Helsinki University of Technology
Department of Electrical and Communications Engineering
Networking Laboratory

Olli-Pekka Lamminen

Implementation and Performance Analysis of a Delay Based Packet Scheduling Algorithm for an Embedded Open Source Router

Master's thesis submitted in partial fulfillment on the requirements for the degree of Master of Science in Technology
Espoo, 6 June 2007

Supervisor: Professor Raimo Kantola

Instructor: Lic.Sc.(Tech.) Marko Luoma

Helsinki University of Technology
Abstract of the Master's Thesis

Author:	Olli-Pekka Lamminen
Title:	Implementation and Performance Analysis of a Delay Based Packet Scheduling Algorithm for an Embedded Open Source Router
Date:	June 6, 2007 Number of pages: 9+80
Department:	Department of Electrical and Communications Engineering
Professorship:	Networking Technology
Supervisor:	Professor Raimo Kantola
Instructor:	Marko Luoma, Lic.Sc.(Tech.)

Appliances used to control and convey network traffic are staring to resemble more and more full grown computer systems. Many of the new appliances start as a combination of standardized parts running custom software on an freely available embedded operating system. Linux operating system has gained a foothold in embedded systems market. Its extensive hardware support and availablility of the source code have made it a natural choice amongst multiple networking hardware vendors.

The purpose of this thesis is to implement a delay-bounded hybrid proportional delay packet scheduler for Linux kernel in an embedded environment, and to validate the scheduler's functionality against a selection of other packet schedulers. The implementation is created as an extension to the Linux traffic control system. The new code uses already existing application programming interfaces and is written following the community established kernel programming guidelines. The validation is done by comparing both theoretical and practical performance of the new scheduler against first-in, first-out, priority queuing and class based queuing schedulers.

The major conclusions drawn from this work are that the open source operating systems like Linux are suitable for embedded development projects, and that the hybrid proportional delay packet scheduling algorithm does not put any more strain on the hardware than the other packet schedulers tested. Any future work on subject can be performed on standard hardware with current versions of available open source operating systems.

Keywords:	hybrid proportional delay, Linux traffic control, scheduling, scheduling algorithms

Teknillinen korkeakoulu

Diplomityön tiivistelmä

Tekijä:	Olli-Pekka Lamminen
Työn nimi:	Viivepohjaisen pakettiaikataulutusalgoritmin toteutus ja suorituskyvyn mittaus sulautetussa avoimen lähdekoodin reitittimessä
Päivämäärä:	6.6.2007 Sivumäärä: 9+80
Osasto:	Sähkö- ja tietoliikennetekniikan osasto
Professuuri:	Tietoverkkotekniikka
Työn valvoja:	professori Raimo Kantola
Työn ohjaaja:	TkL Marko Luoma

Tietoverkkojen hallinnassa ja liikenteen ohjauksessa käytettävät laitteet ovat alkaneet muistuttaa toiminnallisuudeltaan yhä enenevässä määrin yleiskäyttöisiä tietokoneita. Uudet, markkinoille tulevat laitteet pohjautuvat usein valmiisiin komponentteihin ja avoimen lähdekoodin käyttöjärjestelmään, joiden päälle on koottu räätälöityjä ohjaussovelluksia. Kattavan laitteistotuen ja avoimen lähdekoodinsa ansiosta Linux-käyttöjärjestelmä on valittu usean sulautetun verkkolaitteiston pohjaksi.

Tämän työn tarkoitus on toteuttaa viiverajallinen suhteutettua hybridiviivettä käyttävä pakettiaikataulutin sulautetussa ympäristössä suoritettavaan Linux-käyttöjärjestelmäytimeen. Pakettiaikatauluttimen toiminnallisuus varmennetaan vertailemalla sitä kolmeen muuhun pakettiaikatauluttimeen: yksinkertaiseen jonoon, prioriteettijonoihin ja luokkapohjaiseen jonotukseen. Aikatauluttimen toteutus käyttöjärjestelmäytimeen noudattaa valmiita rajapintoja ja ytimessä käytettävää ohjelmointitapaa. Toiminnallisuuden varmentamisessa käytetään sekä teoreettisia että käytännönsuorituskykyä mittaavia menetelmiä.

Työn tuloksena todetaan avoimen lähdekoodin käyttöjärjestelmien soveltuvan tämän kaltaisiin kehitysprojekteihin erittäin hyvin. Lisäksi todetaan, että suhteutettua hybridiviivettä käyttävä aikataulutusalgoritmi ei kuormita laitteistoa sen enempää kuin mikään muukaan testattu aikataulutusalgoritmi. Lisäksi todetaan standardoidun laitteiston ja ajantasaisten avoimen lähdekoodin käyttöjärjestelmien soveltuvuus käytettäväksi myös tulevissa aiheeseen liittyvissä tutkimusprojekteissa.

Avainsanat:	aikataulutus, aikataulutusalgoritmit, liikenteen kontrollointi Linuxissa, suhteutettu hybridiviive

Preface

This thesis was funded by the Helsinki University of Technology Networking Laboratory as a part of the TIEVA project. The work is a continuation of the IRoNet project in its aim to create adaptive policy-based networks.

I would like to thank my supervisor, professor Raimo Kantola, and my instructor Marko Luoma for their guidance. I would also like to thank Mika Ilvesmäki for his comments, and the staff of the Networking Laboratory for providing a positive work environment. I also give my thanks to my friends and family for their support during my studies.

Espoo, 6 June 2007

Olli-Pekka Lamminen

Contents

Abbreviations

ALTQ	ALTernate Queueing
API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BSD	Berkeley Software Distribution
CBQ	Class Based Queuing
CPM	Communications Processor Module
CPU	Central Processing Unit
DBHPD	Delay-Bounded HPD
DDP	Delay Differentiation Parameter
DIMM	Dual In-line Memory Module
DMA	Direct Memory Access
DNS	Domain Name Server
DSCP	Differentiated Services Code Point
E-mail	Electronic mail
EWMA	Exponentially Weighted Moving Average
EWMA-pe	EWMA based on Proportional Error of the estimate

EWMA-r	EWMA with Restart
FIFO	First In, First Out
FSR	Frame Synchronized Ring
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
HFSC	Hierarchical Fair Service Curve
HPD	Hybrid Proportional Delay
HTB	Hierarchical Token Buckets
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IRQ	Interrupt ReQuest
LAN	Local Area Network
MAC	Media Access Control
NIC	Network Interface Card
NMS	Necsom Media Switch
NVRAM	Non-Volatile RAM
PRIQ	PRIority Queuing
RAM	Random Access Memory
SDRAM	Synchronous Dynamic RAM
SIP	Session Initiation Protocol
TBF	Token Bucket Filter
TC	Traffic Control

TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTL	Time To Live
UDP	User Datagram Protocol
VoIP	Voice over IP
WWW	World Wide Web

Chapter 1

Introduction

1.1 Background

A computer network can be described as a bunch of computers interconnected by machines specialized in conveying data packets. These specialized machines – routers, gateways, firewalls – are designed to ensure that the flow of traffic from one computer to another reaches its destination unhindered and as effectively as possible. Originally the service in networks like the Internet was based on a best effort model. The traffic originating from one machine was sent along its destined route with a hope that it would not encounter any major difficulties along the way.

Modern networks have evolved beyond the limitations of the best effort model. Not only has the volume of traffic grown tremendously, but the variety of applications using the network has become more diverse. When at one point it was enough to be able to transmit bulk data between two computers overnight, now a network is required to transmit data fast, stably and reliably.

Applications like voice calls, video conferencing, and multiplayer games require connections with low delays and minimal delay jitter. At the same time other applications like web browsing and p2p transfers use the same network to move vast amounts of data around. Some of these applications, like p2p, behave aggressively and try to use as much bandwidth as possible. Others like voice and video calls require a certain minimum amount of

bandwidth to be available in order to function properly.

Several service quality increasing schemes have been devised to balance the situation. Firewalls and gateways have been designed to block and shape unwanted traffic to ensure more bandwidth for wanted traffic. Routers on network edge can be used to classify traffic and prioritize traffic classes to ensure better service to more critical classes. The classes are weighed and serviced in order determined by a packet scheduler. The scheduler uses a scheduling algorithm to determine the service order of the packets based on some prespecified criteria.

Most of the currently used scheduling algorithms are based on either strict priorities, like priority queuing, or bandwidth allocations, like class based queuing. Neither of these options feel natural if the quality that needs to be ensured is specified in terms of delay and delay variations. Hybrid proportional delay scheduling was developed to fill this hole within the currently used scheduling algorithms [8].

HPD was designed to prioritize traffic based on the delays packets experience in a network router. Later extensions to the HPD algorithm have introduced alternative methods for delay calculations and hard delay limits to ensure faster service times for the most delay intolerant traffic classes.

Machines controlling the flow of traffic and quality of service in modern networks vary from standard PCs managing connections in homes and small offices to specialized hardware and software running in corporate data centers and backbone networks of large service operators. In between the standard PCs and specialized hardware running custom software are several embedded solutions using customized bits of hardware with freely available open source operating systems.

These embedded solutions have found their place amongst small businesses that might have neither monetary nor technological resources to design fully customized systems. The ability to choose one bit of hardware from vendor A and combine it with another bit of hardware from vendor B both supported by a freely available operating system has made it possible for newcomers to come up with innovative and succesful products.

One of the most common operating systems for such products is Linux. The reasons for Linux's success are in the free licencing policy and large community support for both hardware and additional software products. The operating system itself has been ported on various architectures and supports a wide range of different hardware devices. The common programming interfaces have made it easy to port additional bits of software between the opearting system's supported architectures. All of this put together has lowered the threshold for newcomers to enter the market.

1.2 Objective of the Study

The purpose of this work is to implement the delay-bounded HPD packet scheduling algorithm for Necsom Media Switch embedded router hardware. The hardware uses a Linux 2.4 kernel and is able to run traffic control and routing applications. The implementation will follow the existing guidelines in the Linux kernel and will complement the other packet scheduling algorithms already implemented in Linux.

The validation of the implemented algorithm is performed by measuring both theoretical and practical performance of the implementation. These measurements are compared against the performance of three other scheduling algorithms, FIFO, PRIQ, and CBQ. The effectiveness and success of the implementation can then be based on these comparisons. Further research on the suitability of both the HPD algorithm and the NMS embedded environment can then be based on the results of this thesis.

1.3 Structure of the Thesis

The Second Chapter describes IP networking in traffic control Linux operating system. Both the receiving and transmitting sides of the Linux networking kernel are described in detail, as is the way Linux traffic control functions and connects to the rest of the networking kernel.

Chapter Three gives an overview of packet scheduling algorithms and concentrates on four major algorithms in particular: FIFO, PRIQ, CBQ and HPD. The functionality of these four scheduling algorithms is discussed

in quite some detail. With the HPD algorithm the functionality of three different EWMA algorithms is explained.

The Fourth Chapter describes the implementation process of the HPD algorithm for NMS hardware. It begins with a detail description of the NMS hardware and continues by describing the relevant APIs for a new scheduling discipline. The actual implementation is presented in detail starting with the data structures and ending with descriptions of the different routines and algorithms the HPD scheduling discipline uses.

Fifth, Sixth and Seventh Chapters focus on validating the implementation. The fifth chapter talks about how the functionality of the implementation can be measured and presents the measurement scenarios used. In the sixth and seventh chapters the results of the measurements are revealed and discussed.

This thesis ends with concluding thoughts about the suitability of the Linux operating system and the used embedded hardware for the purposes of presented work. The Final Chapter also includes further discussion about the validation results and suggests a few outlines a future research on the subject should take.

Chapter 2

Linux and Networking

Linux is a Unix-like open source operating system well suited for networked environments. Linux was born as a hobby project in early 1990's and has since grown in popularity, especially in network servers and embedded systems [18, 21]. According to market research company IDC, Linux achieved 25% market share in server operating systems in 2004 [26].

Linux features a monolithic kernel. The networking subsystem is an integral part of this kernel and includes facilities for firewalling, forwarding and traffic shaping among others. Altough being monolithic, the kernel is also built modular, which makes it easier for the developers to include to or exclude features from the kernel.

Other factors contributing to Linux's success on embedded platforms are the free nature of the source code, portability to various different architectures, availability of developement tools and driver support for a wide variety of different devices [18, 21].

2.1 Linux Kernel Networking Subsystem

The networking subsystem in the Linux kernel is dividable into four different parts. First part is the networking core, which includes protocol independent code, like generic datagram handling routines, data structures and device interfaces. The second part includes protocol specific code required for handling different types of networks like Ethernet, IP, ATM and X.25.

The third part is the network scheduler, which handles traffic prioritizing and shaping. The last part includes network drivers to accommodate all the different bits and pieces of supported networking hardware [14].

This thesis concentrates on the IP version 4 networking, and the packet scheduling codes only. The information in this thesis is relevant to Linux kernel version 2.4. In kernel version 2.6 the networking API has been revised to achieve better performance under higher workloads.

2.2 IP Networking and Packet Forwarding

When we look at the kernel networking subsystem from the forwaring point of view, we can divide the functionality into three different layers: lowest or hardware layer, middle or network layer and upper or transport layer [20]. These layers are illustrated in figure 2.1. The lowest layer is tasked with moving packet data between network interface hardware and kernel memory space. This involves answering and scheduling interrupts from networking hardware and moving the packet data between kernel memory space and memory accessible by the networking hardware.

The network layer handles Ethernet and IP level processing. The functions in this layer perform MAC address resolving, IP address resolving, and forwarding, shaping and queuing tasks. The kernel maintains an ARP table for mapping MAC addresses in the local network to IP addresses and vice versa. If an incoming packet is not destined to any of the local IP addresses then a kernel routing table is checked to find out the next hop destination towards the packet's intended destination. Traffic shaping and queuing are performed for outgoing packets. Normally any outgoing packets on an interface are put in a single queue to wait when the outgoing interface can send them out. If a more complex queuing schema is used, there can be multiple queues with different priorities or prioritizing algorithms per interface. With shaping the traffic put into these queues can be further regulated to better match the required quality levels.

The transport layer handles ICMP communications and IP subprotocol level processing, including handling TCP connection states and separating data for delivery between kernel and userland programs.

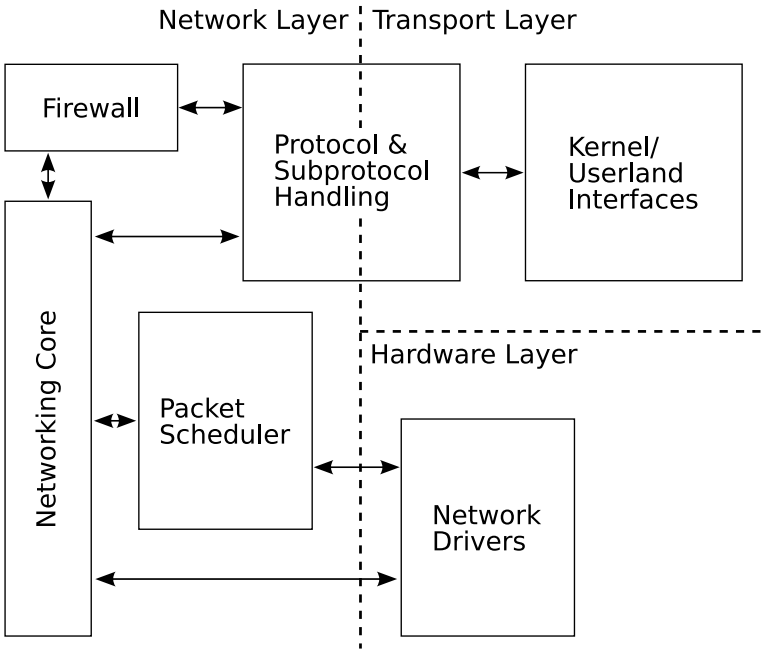

Figure 2.1: Architecture of Linux kernel networking subsystem

Interlayer communication is handled with API calls and the data is transmitted inside data structures defined in these APIs. There are two main data structures used within the networking subsystem. Struct `sk_buff` is used for holding the packet's information, including all the headers and data. Struct `sock` stores information of network and Unix sockets used when conveying data between kernel and listening userland applications [20, 11].

2.2.1 Receiving traffic

When a packet arrives from a network to a machine running Linux, it is first interpreted by a network interface card (NIC). NIC firmware copies the incoming packet data to received data ring buffer, located in a memory area shared between the networking hardware and kernel, using direct memory access (DMA). After this the networking hardware sends an interrupt request (IRQ) to the central processing unit (CPU) [20]. This is shown in figure 2.2.

After receiving the IRQ, the CPU jumps to network driver code as in-

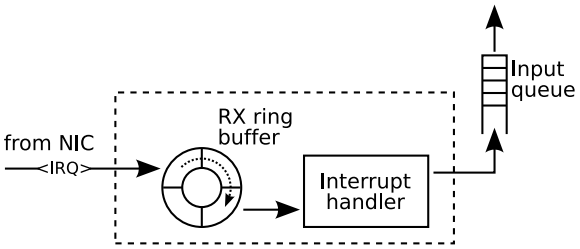

Figure 2.2: Lower layer receiving

structed by the kernel. The network driver copies and formats the incoming data to an `sk_buff` data structure and moves it to an incoming traffic queue. Then the driver assigns a softIRQ, which is not an IRQ for the CPU, but a programmable IRQ for the kernel scheduler [20, 29]. When the kernel scheduler receives the assigned softIRQ, it starts executing a special softIRQ handler, which checks the incoming packet queue and moves pending packets one by one for further processing in network layer functions, namely `ip_rcv()`.

Upon arriving to network layer, the packet is first processed by `ip_rcv()` function as shown in figure 2.3. This function checks the packet's destination address, and depending on the packet's destination and system configuration, does one of the following three things: if the packet's destination address is one of the addresses of the local machine, the packet is delivered to transport layer (local delivery in figure 2.3). If the packet's destination is some other host and packet forwarding is disabled in the kernel, the packet is dropped. If packet forwarding is enabled and the packet's destination is some other host, this machine acts as a router and `ip_forward()` function is called for further processing [20, 11].

`Ip_forward()` function analyzes the packet and resolves its next-hop destination. First the packet's time-to-live value (TTL) is checked. If the TTL is one or less, an ICMP Time Exceeded message is sent back to sender and the packet is dropped. Otherwise a next-hop route is resolved by checking packet's destination address against kernel's routing table. Firewall policies are checked for whether or not forwarding the packet is allowed. If the next-hop destination cannot be found, or a firewall policy denies forwarding the

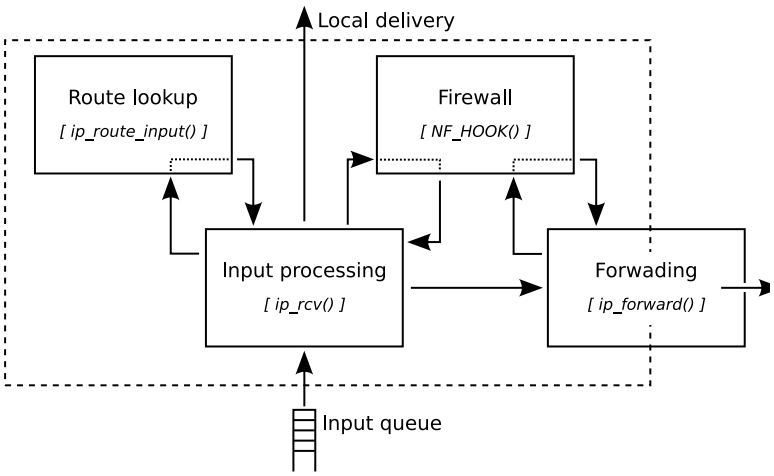

Figure 2.3: Network layer receiving

packet, an ICMP Destination Unreachable message is sent back to sender.

After the next-hop destination is resolved, and permission to send is acquired from the kernel firewall, the system is ready to send the packet forward. At this point, if the system is configured to send ICMP Host Redirect messages, one is generated and sent back to sender. The packet's TTL value is decreased and the packet data is copied to a new `skb_buff` structure with the new destination information. This structure is then used for transmission as described in the next section [19].

If the packet was destined to one of the local addresses, the transport layer functions are used to extract the packet data for corresponding user-land listening application. In case of TCP traffic, transport layer functions also keep track of TCP connection state, acknowledgements, duplicates, and ordering [20, 11]. More profound analysis on transport layer's functions and tasks is not relevant from the forwarding point of view and is thus outside the scope of this thesis.

2.2.2 Transmitting traffic

Transmission, illustrated in figure 2.4, is initiated either from transport layer via `ip_output()` function or from forwarding function via `ip_send()` function. Both of these functions call `ip_finish_output()`, which in turn deliv-

ers the packet for queuing on an outgoing interface. If a packet scheduler is configured for the outgoing interface, the scheduler's `_enqueue()` function assigns the packet to the right queue. If a packet scheduler is not used, then all the outgoing packets are placed in a single queue to wait for transmission. The packet scheduler runs independent of the rest of the networking code and its functionality is further explained in the next section [20].

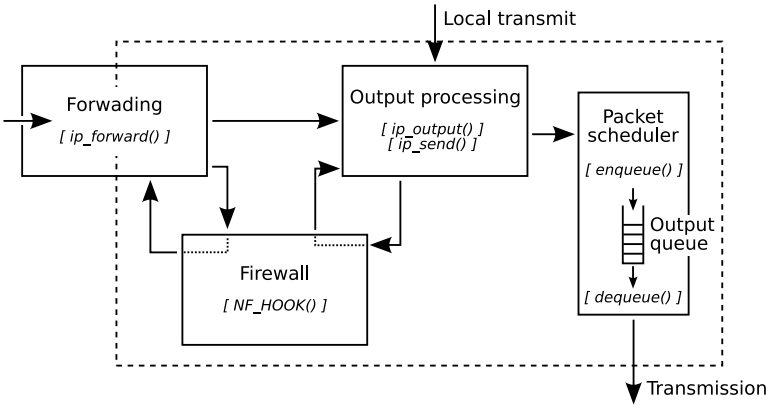

Figure 2.4: Network layer transmission

Whether a multi-queue scheduler or a single queue is used, when a packet is dequeued, outgoing NIC's device driver is called to format the packet data for line transmission, and copy it to an outgoing data ring buffer, shown in figure 2.5. Like received data ring buffer, this buffer too recides in a memory area accessible both by kernel and the NIC hardware by DMA. After the packet data is copied to the ring buffer, the driver tells the NIC that new data is ready to be sent out. After the NIC has sent the data out, it tells the CPU about this. The CPU then calls the device driver to process this information and to schedule a softIRQ for the kernel to clear the memory used by the packet. Depending on the NIC and the driver, this may happen after either a single packet or a group of packets has been sent out [20].

2.3 Traffic Control

Traffic control has been a part of the stable distribution of the Linux kernel since the version 2.2.0 [5]. Traffic control provides means to prioritize and

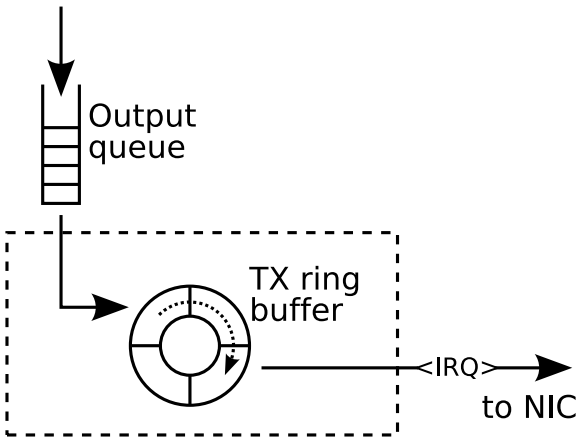

Figure 2.5: Lower layer transmission

limit traffic originating from or relayed by a Linux machine.

Traffic control hooks on the output side of the network layer part of the networking subsystem, just before outgoing packets are transmitted to hardware layer. Traffic control consists of three functionally different parts; queuing disciplines, filters and classes. The traffic control has been designed to be stackable and modular. This allows the use of multiple different traffic control schemes in conjunction with each other.

Queuing disciplines are the basic building blocks of a traffic control system. A queuing discipline includes zero or more classes to prioritize traffic and zero or more filters to direct traffic into these classes. Every class can contain an additional queuing discipline with its own classes and filters [1]. An illustration of queuing discipline and its parts is shown in figure 2.6.

2.3.1 Queuing disciplines

Queuing disciplines come in two flavours. Classless queuing disciplines can be used to limit the overall traffic but are not usable for prioritizing traffic. By default, a classless FIFO discipline is assigned for every interface. FIFO discipline is also used as an initial child discipline for every class in classful disciplines.

Classful queuing disciplines are more versatile. In addition to limiting traffic they can also be used for prioritizing it. A classful discipline includes

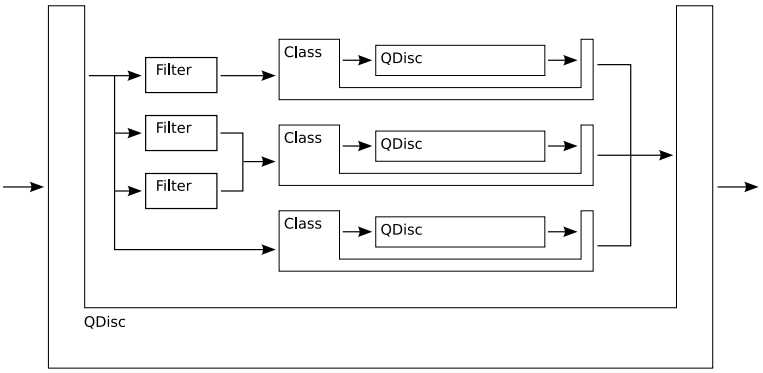

Figure 2.6: Queuing discipline building blocks

a scheduler, one or more classes and any amount of filters linked to these classes [1].

When a packet arrives for queuing, the `_enqueue()` function of a discipline selects in which class the packet is queued to based on the filters configured in the discipline. When the kernel decides it is time to dequeue packets on an interface, the discipline's scheduler, evoked with `_dequeue()` function, selects a class from which a packet is dequeued from. The scheduler can make its decision based on various criteria like static priorities, queuing delays, or used traffic quota.

2.3.2 Classes

Classes are used to differentiate traffic. Each class includes its own queue in form of an additional queuing discipline. These disciplines are by default FIFO disciplines, but for example a token bucket filter discipline can be used to limit the rate of traffic per class.

Each class has its own set of class variables. These variables are used by the discipline queuing functions to make scheduling decisions. Class variables can hold any information available to class, such as byte and packet counters, delay values or differentiation parameters.

2.3.3 Filters

Filters are used to assign traffic to classes. A filter can evaluate any information available from an incoming packet as well as any information available for a class or a queuing discipline. Filters can also be stateful and thus use historical information as a base of filtering decisions. Furthermore, filters can be chained in ways that allow for more versatile filtering schemes.

2.3.4 Policing

Policing is used to bound traffic within specified limitations. There exists no particular location for making policing decisions. Policing rules can be implemented in any part of traffic control. Some queuing disciplines, like Class Based Queuing (CBQ), include their own policing components on queuing discipline and class levels. In this case the policing is done in discipline's or class's `_enqueue()` function. There are also filters, like Token Bucket Filter (TBF), which can be used to perform policing.

2.3.5 Management Applications

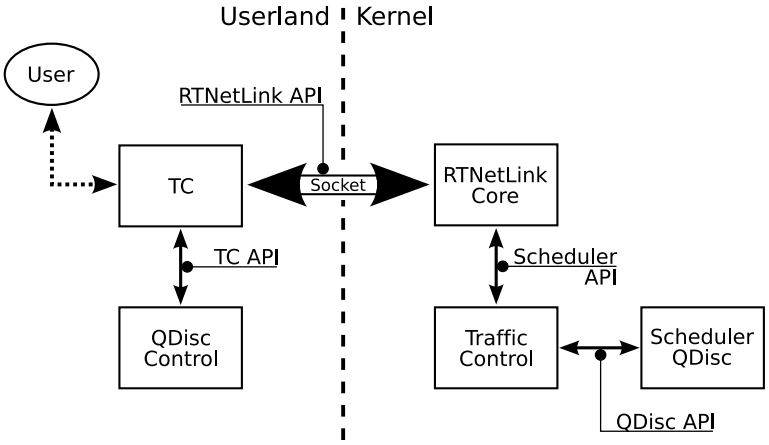

Figure 2.7: Call paths and APIs

Traffic control management tools are included in *iproute2* software package. The program used for managing traffic control is called *tc*. *Tc* uses RTNetLink API, which extends NetLink API, for communication between

userland and kernel facilities. This API includes commands to access kernel forwarding and packet scheduling subsystems, and is used also to manage kernel routing tables [13]. This and other APIs used with traffic control are depicted in figure 2.7.

Like *iptables*, *route* and similar Linux networking management utilities *tc* does not use a configuration file. Instead, *tc* is used by executing multiple successive program calls with varying command line parameters. *Tc* can be used for adding, changing, and replacing queuing disciplines, classes, and filters. It can also be used to show the status of a discipline, a class or a filter [13, 16].

Chapter 3

Packet Scheduling Algorithms

IP traffic consists of packets. Packets can be grouped and classified based on similar characteristics they share. Valid characteristics can include source and destination addresses, IP subprotocol, such as TCP or UDP, and source and destination ports if applicable. Packet scheduling algorithms are used to prioritize these traffic classes. Traffic prioritization can be used e.g. to ensure minimum service levels for certain customers or applications, or limit unwanted traffic in a network.

3.1 Applications

Different services and applications require different qualities from the network. Some services do not require fast response times but can easily use any amount of free bandwidth available. Examples of such services are WWW browsing, FTP transfers, and e-mail. Other services, such as VoIP and video conferencing, are much less tolerant to delay variations and require both bandwidth and delay guarantees. On some services, like fast-paced online games, low latencies can become crucial for achieving the advertised user experience.

3.2 Algorithms

Basic scheduling procedure is to first classify incoming packets and then send packets out based on the priorities assigned to classes. Usually classes with higher priority get precedence over others. Prioritization is done by a scheduling algorithm and can be based on multiple criteria, like fixed priorities, bandwidth or delay.

Scheduling algorithms can be divided to static and dynamic algorithms. Static algorithms have fixed priorities assigned to classes and always prefer one class over another. One example of a static algorithm is first in, first out (FIFO). FIFO has only a single queue, and the packet which arrived first also gets sent out first. Another example is priority queuing (PRIQ), in which traffic is classified, and classes are processed in strict priority order.

Dynamic scheduling algorithms prioritize classified traffic according to varying criteria such as bandwidth and delay. Examples of dynamic algorithms are class based queuing (CBQ), hierarchical fair service curve (HFSC), hierarchical token buckets (HTB), and hybrid proportional delay (HPD).

In CBQ, HFSC, and HTB classes are organized in a hierarchical tree and each class is assigned a bandwidth allotment from the total bandwidth available for the discipline. Classes running close or over the maximum of their allotment can borrow bandwidth from their parents and siblings. The main difference between CBQ and HTB is that HTB uses token buckets for estimating bandwidth rates while CBQ uses exponentially weighted moving average of idle time [9, 7]. In HFSC service grade given to leaf classes varies as a function of classes' delays [24].

HPD differs from the previous in that classes are not organized in a tree-like hierarchy. Instead classes are considered as peers, just like in priority queuing. The classes' priorities are dynamical and calculated as a function of current and historical delays. Historical delay is approximated with an exponentially weighted moving average (EWMA) estimator [8, 3].

3.2.1 First In, First Out

First in, first out (FIFO) is a basic queuing algorithm often referred as no queuing at all. FIFO has only one queue and no classes. Packets get sent out in the order in which they arrive without prioritization. This can be achieved either by linked list or a ring buffer or with a hash table indexed by packet arrival time values. The latter method is used in many appliances based on specifically designed integrated circuits while the two former methods are more common in operating system kernels like Linux or BSD. FIFO is a natural choice anywhere queuing without any prioritization is needed. As such it is used as a default, or only available queuing algorithm in many network appliances. The quality of a FIFO queue can be controlled by altering the length of the queue. Longer queues mean less dropped packets but also longer service delays. An example of a FIFO queue is shown in figure 3.1.

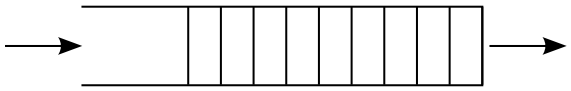

Figure 3.1: FIFO queue

3.2.2 Priority Queuing

Priority queuing (PRIQ) is a basic classful scheduling algorithm. It consists of multiple classes with static priorities. This is usually achieved either with each class having its own, usually a FIFO queue, or with a single sorted queue with higher priority packets added to the front or middle rather than to the end of the queue. The first approach is the one used in Linux kernel and is illustrated in figure 3.2. A single queue solution can be implemented with e.g. a linked list or a hash table indexed by class and arrival time.

Incoming packets are classified to one of the classes. Outgoing packets are selected based on classes' priorities starting from highest and proceeding downwards. Higher priority class's queue gets emptied in full before moving to lower priority one. This makes it possible for a lower priority class to suffer from starvation if there is a continuous influx of packet to higher

priority classes. In order to avoid starvation of critical traffic, such traffic must always be assigned to the highest priority class.

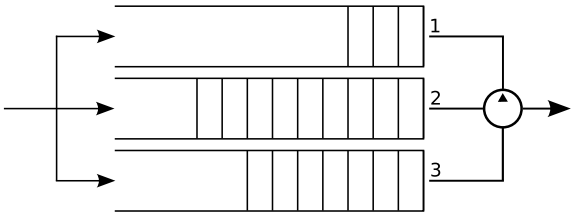

Figure 3.2: Priority queuing

The possibility of starvation makes PRIQ unsuitable for networking scenarios where there exists multiple classes of traffic, which need to be served, but which should also have different priorities. Starvation can also happen if the link used with PRIQ is very slow, e.g. a modem line. In such scenarios it is possible for a single TCP connection to consume the resources of the whole link if the packet size is large enough and the interarrival time of the packets is short. PRIQ is applicable mostly in scenarios with only a few traffic classes, when some of the classes have to be served even at the cost of other traffic, and the traffic from these classes is guaranteed not to saturate the available link capacity.

An example of such a scenario is a corporate LAN where phone calls are made with VoIP technology through the same network as regular data uses, and the available bandwidth is greater than bandwidth required for VoIP. In such a network PRIQ would allow VoIP to function even when other types of data traffic, such as web surfing or file transfers, would otherwise use all the available network capacity.

3.2.3 Class Based Queueing

Class based queuing (CBQ) is a classful scheduling algorithm where classes form a tree-like hierarchy. Starting from the root class each class can have zero or more children. Each child class has its own bandwidth allotment taken from the allotment of the parent class. This way the sum of bandwidth allocated amongst all the leaf classes can equal but cannot exceed the bandwidth available for the root class [9]. An example of CBQ class

hierarchy is given in figure 3.3.

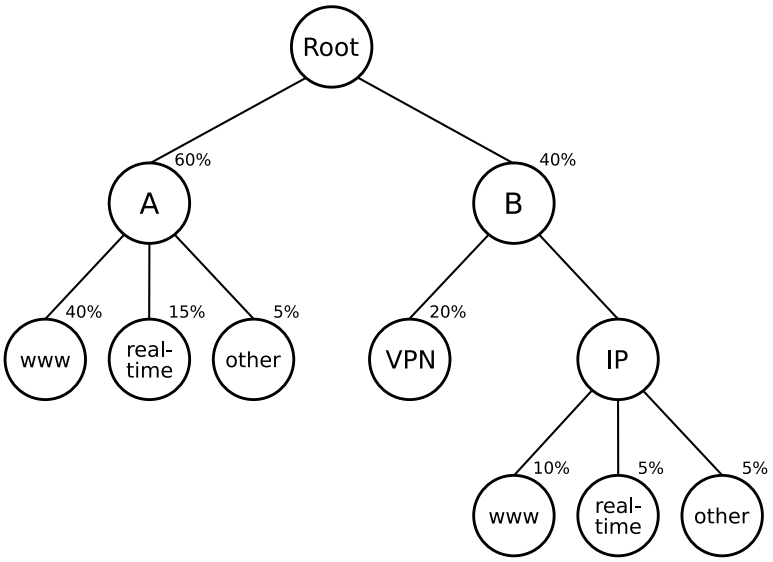

Figure 3.3: Tree-like structure of CBQ classes

CBQ introduces a concept called "borrowing". "Borrowing" means that a class running out of its allocated bandwidth can use bandwidth allocated to some other class as long as that class has not ran out of its allocated bandwidth. An example of borrowing is given in figure 3.4. To control borrowing, CBQ has three special class attributes: bounded, isolated and exempt [9].

By default all classes can borrow bandwidth from their parents and siblings. Bounded classes are not allowed to borrow bandwidth from their parents or siblings. This means that a bounded class can never transmit more traffic than its allocated bandwidth allows it to. Isolated classes behave like bounded classes and additionally do not borrow bandwidth to their siblings. This means that bandwidth assigned for an isolated class is usable only by that class and its children. Exempt classes are never restricted from borrowing bandwidth from other classes no matter what the bandwidth usage of the target class.

CBQ offers certain benfits over PRIQ. Bandwidth limits assigned per class ensure a base level of quality and reduce the possibility of starving. Borrowing allows classes experiencing sporadic traffic spikes to acquire more

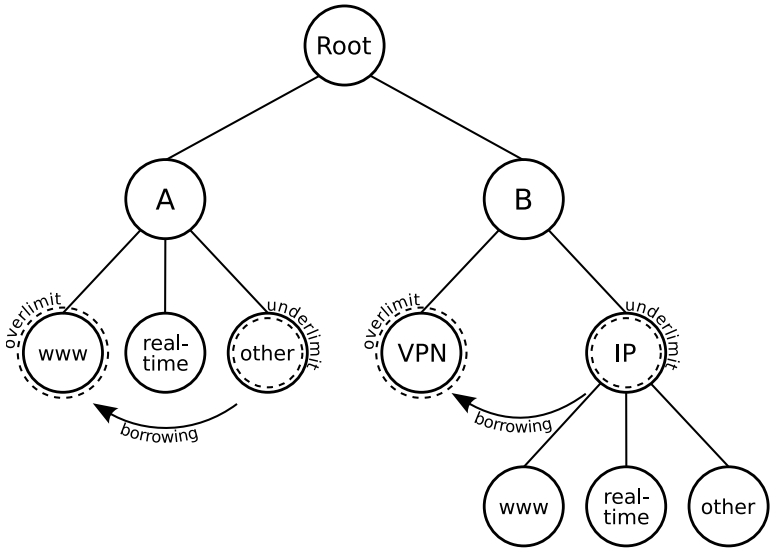

Figure 3.4: Example of borrowing in CBQ

resources if other classes have resources available. Isolation guarantees available bandwidth even if other classes would like to use more resources than there are available for them. These properties make CBQ more versatile than PRIQ for large networks with multiple traffic classes with different service requirements.

CBQ also has some disadvantages. Some CBQ functionalities, especially borrowing, require a lot of heavy calculations which can strain the router hardware. Some calculations can be avoided by bounding and isolating classes and thus limiting the number of borrowing calculations. However, these qualifiers are often applicable for only a few classes, and with isolation the resources allocated for an isolated class become unusable by its sibling classes, which in turn causes unefficient use of link capacity. If isolated classes receive little or no traffic, and other classes are running out of their assigned resources, some traffic will be dropped, even if the link is not running close to its optimal capacity.

3.2.4 Hybrid Proportional Delay Queueing

Hybrid proportional delay queuing (HPD) is a classful packet scheduling algorithm in which class priorities are calculated at run-time based on the normalized hybrid delay values of the classes [8]. Like in PRIQ, the class with the highest priority gets served first. Because the delays, and thus priorities, grow if a class does not get service, possibility of starvation is minimized. HPD class structure is shown in figure 3.5.

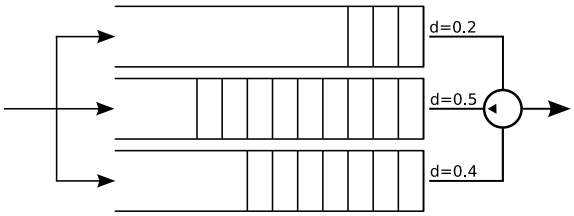

Figure 3.5: HPD queuing

A variant of the HPD algorithm, delay-bounded HPD (DBHPD), introduces a hard delay bound for one of the classes in the scheduler [3]. If a class with a delay bound is about to violate these bounds, it gets immediate service. This functionality helps assuring delay bounds for traffic classes less tolerant of high delays or delay variations. Delay bounds might, however, also cause starvation. If the bound is set too tight, and the class with the bound is receiving more traffic than the hardware can process within the timeframe of the bound, other classes will not get any service at all.

HPD is a valid solution for prioritizing delay dependent traffic over other traffic types. Delay bounding allows setting an upper limit for the class's delay. This is useful for voice and video communications, as well as for other types of real-time traffic where high latencies can degrade user experience significantly. Delay based prioritization also helps to decrease delay variations by stabilizing delays of the classes on a loaded, but not full, link.

In HPD a class's priority is calculated from its normalized hybrid delay value. This value, h, is infcluenced by both short and long term queuing delays, and is calculated with

$$h = g\tilde{d}_i(m) + (1-g)\tilde{w}_i(m), \tag{3.1}$$

where $\tilde{d}_i(m)$ is the normalized average queuing delay, $\tilde{w}_i(m)$ is the normalized head waiting time and g is the weighting factor.

Normalized average queuing delay is calculated from the average queuing delay $\bar{d}_i(m)$ with

$$\tilde{d}_i = \frac{\bar{d}_i}{\delta_i}, \tag{3.2}$$

where δ_i is a class dependent weighting factor called delay differentiation parameter (DDP).

Average queuing delay is estimated with an exponentially weighted moving average (EWMA) estimator. In addition to basic EWMA estimator two other EWMA estimators have been proposed by Antila & Luoma [2]. All of these estimators use current and historical delays as a base of their calculations.

Simple EWMA Estimator

Simple EWMA estimator uses exponential smoothing for calculating the average queuing delay

$$\bar{d}_i(m) = \gamma_i d_i(m) + (1-\gamma_i)\bar{d}_i(m-1), \tag{3.3}$$

where $0 \leq \gamma_i \leq 1$ is a weighting factor between current and historical delay. EWMA is a good option for calculating average delay in a real-time system, because it doesn't require remembering more than the last average value and thus the calculations are considerably quicker than they would be with e.g. a simple sum estimator.

Antila & Luoma suggest calculating weighting factor γ_i as a function of queue length with

$$\gamma_i(q_i) = \frac{1}{N * \sqrt{q_i} * ln(q_i)}, \tag{3.4}$$

where q_i is the physical queue length and N is the number of classes used [2].

They continue by stating that their function is not an analytically derived expression but can be used as a good guideline for setting the values for γ_i.

EWMA Estimator with Restart

EWMA estimator with restart (EWMA-r) introduces a concept of restart to reset the average delay value of a class when a class has been idle for a given period of time. EWMA-r is useful for bursty classes where traffic is not constant and idle periods occur more often than not. With such classes the long term average delay does not give an accurate display of the class's queuing delays and thus should not be taken into account when determining prioritization.

EWMA-r uses two different methods for calculating the average delay and two new variables for controlling these calculations. When a class begins to receive traffic the average delay is calculated as an exact average with a simple sum method. After a given number of packets have passed through the class's queue, EWMA-r switches to using simple EWMA estimator as described in the previous section. When the class has been idle for a given amount of time, the average delay counter is reset, and delay calculations are initialized using the simple sum method when the class is reactivated in the future.

The behaviour of an EWMA-r estimator for a class i is illustrated in the following pseudo-code where $p_{thresh,i}$ denotes the number of packets before switching to simple EWMA, $cycle_i$ denotes the length of a time period before the class is restarted and $qlen_i$ denotes the current number of pending packets in class's queue.

```
;Initialization
p_samples,i := 0, Σ_samples,i := 0, d̄_i := 0, t_idle,i := 0

;Upon packet arrival
if (qlen_i = 0) then
        Δt_idle,i := now()−t_idle,i
        if (Δt_idle,i ≥ cycle_i) then
                p_samples,i := 0, Σ_samples,i := 0
                d̄_i := 0
        fi
fi

;Upon packet departure
if (p_samples,i < p_thresh,i) then
        p_samples,i := p_samples,i + 1
        Σ_samples,i := Σ_samples,i + d_i
        d̄_i := Σ_samples,i/p_samples,i
else
        d̄_i := γ_i * d_i + (1 − γ_i) * d̄_i
fi
if (qlen_i = 0) then
        t_idle,i := now()
fi
```

(3.5)

EWMA Estimator Based on Proportional Error of the Estimate

EWMA estimator based on proportional error of the estimate (EWMA-pe) is another concept presented in [2] for adjusting EWMA estimator. Where EWMA-r uses restart to better cope with bursty traffic EWMA-pe uses a changing multiplier n for γ_i to mitigate effects of rapid changes in queuing delay.

$$\bar{d_i}(m) = n * \gamma_i d_i(m) + (1 - n * \gamma_i)\bar{d_i}(m-1). \tag{3.6}$$

The idea of EWMA-pe is to multiply γ_i with n when the current delay d_i varies considerably from the average delay $\bar{d_i}$. In [2] the value of n is determined from the ratio of $\bar{d_i}$ and d_i as follows:

$$n = \begin{cases} 7, \text{if } 0.4\bar{d}_i(m) > d_i(m) > 1.6\bar{d}_i(m); \\ 6, \text{if } 0.4\bar{d}_i(m) \leq d_i(m) \leq 1.6\bar{d}_i(m); \\ 5, \text{if } 0.5\bar{d}_i(m) \leq d_i(m) \leq 1.5\bar{d}_i(m); \\ 4, \text{if } 0.6\bar{d}_i(m) \leq d_i(m) \leq 1.4\bar{d}_i(m); \\ 3, \text{if } 0.7\bar{d}_i(m) \leq d_i(m) \leq 1.3\bar{d}_i(m); \\ 2, \text{if } 0.8\bar{d}_i(m) \leq d_i(m) \leq 1.2\bar{d}_i(m); \\ 1, \text{if } 0.9\bar{d}_i(m) \leq d_i(m) \leq 1.1\bar{d}_i(m). \end{cases} \tag{3.7}$$

This causes the estimator to become more responsive for the current delay, and behave more agressively, when the queuing delays are erratic and be more passive when the queuing delays are close to constant.

Chapter 4

Implementation

When implementing features to an operating system kernel running on embedded hardware one needs to account for several things. The hardware used in the target appliance can have its own peculiarities as well as performance limitations. There might be only a limited amount of memory available both runtime and for storing the executable program code. The hardware might use its own proprietary buses for internal communication. These buses can require careful timing or be limited in bandwidth. The processing power of the appliance can be limited and a part of it may be reserved for maintaining services required by other parts of the hardware.

The existing codebase gives guidelines for a programmer to follow. A well written existing code follows its own style practices and they should be followed when writing any new code for the system as well. Instead of inventing new interfaces, the existing ones should be used whenever possible to avoid redundancy. When using existing interfaces they need to be implemented fully to avoid problems arising when some part of existing or future code uses them in a currently unexpected way.

Adhering to these guidelines and adjusting to the limitations existing hardware and software pose ensures less problems when finally testing the new implementation. This is important because the testing of software on an embedded hardware can be a time consuming process. In addition to the developement platform being often architecturally different from the target system, there might not be an emulated environment available for testing

and thus the compiled code can only be tested on the target system.

4.1 Router Hardware

The embedded hardware for which this work has been implemented on is called the Necsom Media Switch (NMS). NMS runs a Linux based operating system. The Linux kernel version used by NMS is 2.4.18.

The Necsom Media Switch is built with modular hardware. The main unit of NMS is a connector rack where networking cards can be plugged into. Each networking card includes its own memory banks and a central processing unit. Inside the connector rack the cards are connected via frame synchronized ring (FSR) bus, developed by Technical Research Centre of Finland [23]. A full rack can hold twelve networking cards. If a rack is not full the FSR-bus has to be closed with a shorting piece.

4.1.1 Frame Synchronized Ring Bus

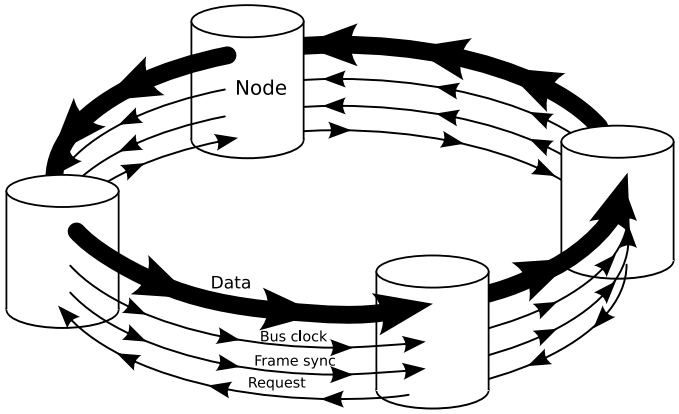

Figure 4.1: FSR bus

Frame synchronized ring bus, depicted in figure 4.1, is a non-blocking, unidirectional ring bus. FSR bus consists of nodes connected by links. A link includes a paraller data bus and three control lines: frame synchronization, request and bus clock [23]. The data bus's width depends on implementation, but usually a bus width of a power of two is used. Frame synchronization line carries synchronization pulses, which indicate the frame

beginnings. The request line is used by the nodes to acquire a permission to transmit. Clock line is used for signal synchronization between the connected devices.

Medium access control is distributed between nodes meaning that every node has to contain the necessary control logic to monitor and access the bus. If an application within a node wants to send data along the bus, it first sends the data packets to the node's interface buffer. The implemented control logic then divides these packets into smaller mini-packets for transmission. Transmission can be done as soon as the node wanting to transmit acquires a turn. During a turn the node 'captures' the first empty frame it sees on the bus and uses it to send its data.

FSR bus used in the Necsom Media Switch is 32 bits wide and runs on 24 MHz frequency. This gives the bus a theoretical maximum throughput of 768 Mb/s.

4.1.2 Networking Card

The networking card used in NMS has four major components: FSR controller, Ethernet controller, CPU and memory. The FSR controller, called X1, handles the connection to the FSR bus backplane connecting all the networking cards in a rack. The Ethernet controller can managee 10/100 Mbit/s Ethernet connection with half- or full-duplex communications. The networking card layout is illustrated in figure 4.2.

The CPU used in the networking card is a Motorola MPC-8260 running at 200 MHz frequency. MPC-8260 is based on PowerPC architecture and includes a communications processor module (CPM), and an embedded G2 core, which is a variant of the 603e microprocessor [10].

The memory on the networking card is divided to two different parts. Runtime memory is provided by a 168-pin SDRAM DIMM. The default amount of runtime memory is 64 MB and the memory can be upgraded to the maximum of 256 MBs. Upgrading the amount of memory requires changes to the card's bootloader code. Non-volatile memory is provided by a 4 MB EEPROM flash chip. This chip stores the bootloader code, kernel, and ramdisk images for the operating system. The flash chip is limited in

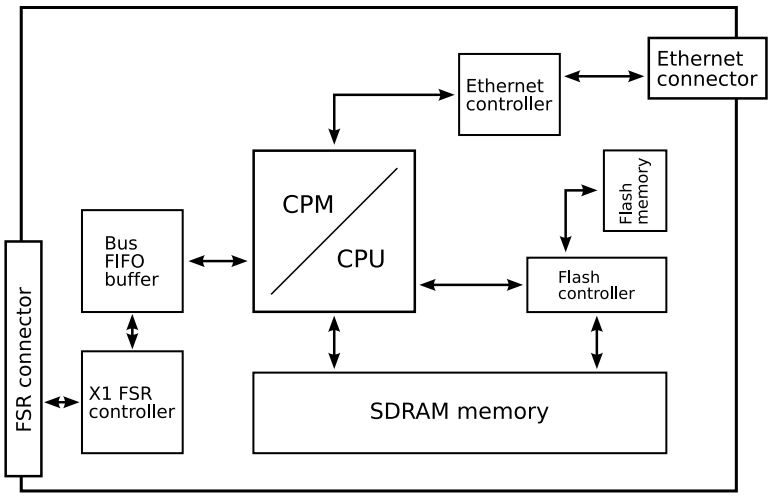

Figure 4.2: NMS networking card

size and after the bootloader code there's only 2 MB of space for the kernel and a compressed ramdisk image. Uncompressed ramdisk is limited to 8 MB in size.

Bootloader is used to start the kernel image and load the ramdisk into the memory. The bootloader also includes facilities for updating the kernel and the ramdisk images via TFTP protocol from a configurable IP address. The first card in the rack is acting as a controller unit for the rest of the cards. Because of this, it cannot be updated directly. To update the first card it first needs to be moved to another position in the rack.

4.1.3 Operating System

The Necsom Media Switch uses Linux version 2.4.18 as its operating system kernel. The kernel code is patched to include drivers for the Ethernet controller and the FSR bus. The FSR bus is accessed from userland via an nrelay-socket. This socket is packet based and benefits from multicast properties provided by the FSR bus. The socket is accessed with the same set of system calls used to access standard UNIX sockets.

Because of the size limitations of the flash memory, the operating system and kernel include only limited functionality. The kernel includes only the necessary drivers and the rest, like packet schedulers, are recommended to

be loaded from modules stored on the ramdisk.

The operating system uses a *busybox* shell, which combines many of the common UNIX utilities in a single small executable. Some other utilities incorporated in the OS are a telnet daemon for receiving outside control connections, module managing utilities and *iptables*. Hardware specific applications like *microcom* and *reset* have been included to access and control other cards in the rack.

Other utilities, like routing daemons, have been ported to the NMS platform. These are not included on the ramdisk by default, but can be included if there exists a need to run them on the NMS. The default ramdisk configuration takes about 2,5 MB of uncompressed disk space and uses 1,2 MB while compressed in the flash memory. The compressed kernel image takes a bit less than 600 kB of space. This leaves about 200 kB of compressed space for additional applications.

The configuration file for IP addresses and routes for the networking cards in a rack is located on a 64 kB NVRAM partition on the first card of the rack. During startup, other networking cards copy the configuration file from the first card.

To avoid conflicts with the interface numbering the interface *eth0* should not be configured in the configuration file. This is because *eth0* always refers to the network interface of the current card. Network interfaces from *eth1* to *ethX*, where X is the number of cards in the rack, should be used. These interfaces match the cards in positions *1* to X of the rack on every networking card in the rack.

4.2 Developement environment

Kernel and application developement for the NMS can be done on any platform and architecture where an appropriate cross-compiler is available. Our chosen developement platform was an x86 based PC running Linux operating system. The main reasons for selecting this environment were the existing documentation for the platform received with the NMS hardware and the ability to test the new kernel features using user-mode Linux kernel patches [27].

Building the developement environment begins with building a cross-compiler for the PPC architecture. For this, I used GNU Compiler Collection (GCC) version 3.3.3. The GNU C Library (GLibC) version I used was 2.3.6. Building and installing a cross-compiler is a pretty straightforward process and is well documented e.g. in [4] and [15].

The Linux kernel version used with NMS is 2.4.18. The base kernel source can be downloaded from kernel.org [19]. Before being able to run the kernel on NMS it has to be patched with nrelay-socket and FSR-bus drivers. This driver patch was provided by Necsom Ltd. The patch affects only lower layer driver code and as such does not add any visible changes to the functionality of the networking subsystem.

The recommended way to add functionality to the kernel is to use loadable modules. This way a problem with the new module does not prevent loading the kernel in memory even though it can still cause the kernel to crash. To be able to use the modules, they need to be included on the ramdisk image. Any additional userland programs necessary for managing added functionality need to be included on the ramdisk as well.

4.3 API Compliance

Three APIs need to be considered when implementing a new packet scheduling algorithm. The first one is the packet scheduling API which is used inside the kernel to manage the scheduler, and enqueue and dequeue packets from the scheduler. The second one is the RTNetLink API which is used between the userland and the kernel to control the scheduler and to request statistics. The third API is the TC API which is used to add support for the new scheduler to the traffic control application *tc*. These APIs are presented in figure 4.3.

4.3.1 Packet Scheduling API

The packet scheduling API has two sets of routines: the queuing discipline operation set and the class operation set. Queuing discipline operation set describes two major and five auxilary routines. The major routines, `_enqueue()` and `_dequeue()`, are used for enqueuing and dequeuing

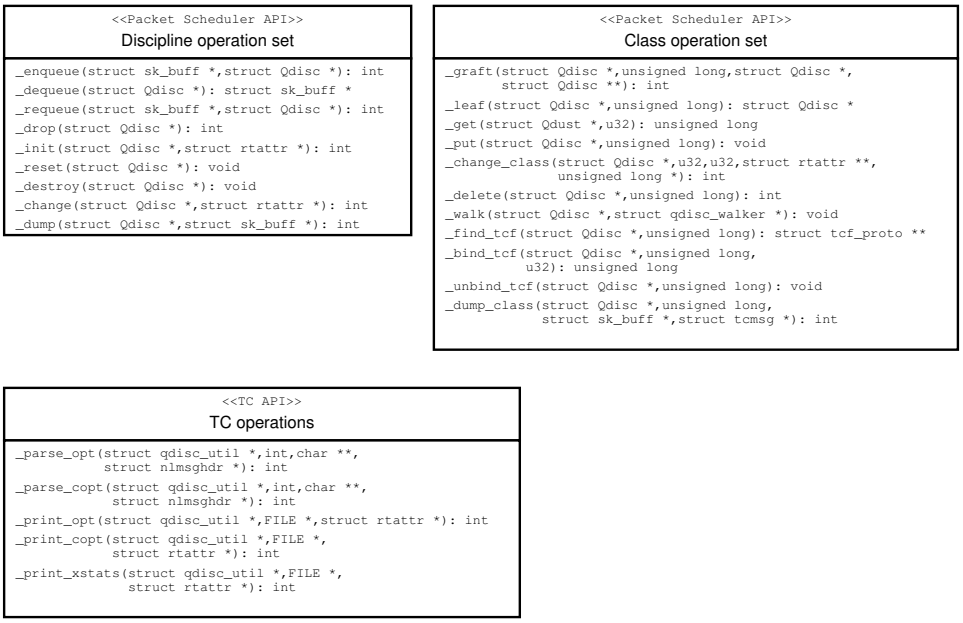

Figure 4.3: APIs at a glance

packets, and are mandatory for every discipline. The auxilary routines are `_requeue()`, `_reset()`, `_init()`, `_destroy()` and `_change()`. Additionally there are two undocumented routines: `_drop()` and `_dump()`.

Class operation set includes routines for creating, deleting and modifying classes as well as manipulating sub-disciplines and filters connected to classes. Other functions included in the class operation set are functions used for retrieving class statistics and a function for executing an external command for each class. The class operation set is mostly undocumented within the kernel source code.

Queuing Discipline Operation Set

Queuing discipline operations can be divided to queuing and management operations. Queuing operations are `_enqueue()`, `_dequeue()`, `_requeue()`, and `_drop()`. `_enqueue()` and `_dequeue()` are used, to pass packets in and out of the queuing discipline. `_requeue()` is used for re-enqueuing once dequeued packets. It is needed with non-standard or buggy devices, which do not dequeue packets properly. `_drop()` is used to drop a single packet

from the discipline.

Management operations are _init(), _reset(), _destroy(), _change(), and _dump(). _init() and _destroy() are used for creating, initializing, and destroying resources used by a queuing discipline. _reset() is used to return the queuing discipline to its initial state. _change() alters the queuing discipline's parameters, and _dump() returns runtime statistics and configuration options from the queuing discipline.

Class Operation Set

Class operations can be divided into class, child queuing discipline and filter manipulation routines. Class manipulation routines include _get(), _put(), _change(), _delete(), _walk(), and _dump(). _get() is used to get a memory reference for a class based on the class identifier, and to keep track of the number of outside references to the class. Memory references are used to speed up class access inside the discipline. _put() is used to inform the class that one of the memory references gotten with _get() is no longer used. If the number of memory references drops to zero the class is removed.

_change() is used to create a new or modify an existing class. Its function differs based on the existence of a requested class identifier. If the identifier does not match with any known class a new class is created, otherwise the matching existing class is modified.

_delete() is used to remove a class. When called, the class is disconnected from the calling queuing discipline. To avoid memory access errors, the class is not physically removed from memory until the memory reference count updated with _get() and _put() routines drops to zero. _delete() counts as a single _put() for the purpose of counting the number of memory references.

_walk() is a multi-purpose routine. When called, it walks through each class belonging to a queuing discipline, counts them, and calls another routine given as a parameter for _walk() with each one of the classes. _walk() is often used with _dump() to retrieve statistics from each class of the discipline. Like its queuing discipline operation counterpart, _dump() is used to retrieve statistics of class usage and its configuration parameters.

Routines used for manipulating child queuing disciplines are `_graft()` and `_leaf()`. `_graft()` is used to attach a new child queuing discipline to a class. `_leaf()` is used to fetch a memory reference to the attached child queuing discipline.

The routines for manipulating filters are `_find_tcf()`, `_bind_tcf()`, and `_unbind_tcf()`. `_find_tcf()` is used to fetch a list of filters belonging to a class. `_bind_tcf()` and `_unbind_tcf()` are used to add to and remove filters from this filter list.

4.3.2 RTNetLink API

RTNetLink API is mainly hidden from a queuing discipline programmer's view. It is used by userland applications to access kernel facilities related to packet scheduling and forwarding. It extends NetLink API, which is used to control sockets. The traffic control application *tc*. *Tc* uses the RTNetLink API to create, change, and remove queuing disciplines, classes, and filters as well as to receive scheduling statistics from the kernel.

4.3.3 TC API

TC API describes routines to use with Linux traffic control application *tc* included in IPRoute2 software package. The API routines are `_parse_qopt()`, `_print_qopt()`, `_parse_copt()`, `_print_copt()`, and `_print_xstats()`.

`_parse_qopt()` and `_parse_copt()` routines are used to parse queuing discipline and class options given as command line parameters. The corresponding `_print_qopt()` and `_print_copt()` routines are used to query these options from kernel and to display them to the user. `_print_xstats()` is used to display additional queuing discipline and class related statistics recieved from the kernel. Many of the existing queuing disciplines use a single routine, usually called `_print_opt()`, for displaying both the discipline and class parameters.

4.4 DBHPD Queuing Discipline

Implementing a queuing discipline begins with the design. The design needs to adhere to the existing APIs and practices followed in the kernel packet scheduler. Major parts of this design are the data structures used with the discipline and the correct use of functions defined in the packet scheduling API.

The data structures need to contain all the information necessary for making scheduling decisions. At minimum a structure for the discipline data is needed. For classful disciplines like DBHPD it is useful to have a class data structure as well. These structures hold both configuration options and the current state the discipline or class are in.

4.4.1 Data Structures

First step in designing data structures is to identify what data should be stored into them. For DBHPD this includes configuration parameters used for calculating the hybrid delay value and variables to store old delay averages used in these calculations. The discipline needs also to know how to refer its classes and the classes might need to form an organized structure between each other.

Discipline Data Structure

The discipline requires to know about two things: its classes, and filters connected to these classes. The packet scheduling API includes a desrciption for a filter list structure, which should be used with every discipline using filters. To use this existing filter list, the discipline structure, shown in figure 4.4, is required to include a pointer to the first filter structure and a counter for the number of filters used. The filters form a singly-linked list that can be indexed with these two variables. Even though the filters do not work correctly in the 2.4.18 kernel, I decided to include this functionality both for future needs and as a good coding practice.

I have decided to store the class references in a single array. The array size is constant and limited to the maximum number of classes. This number

```
struct hpd_sched {
    /* Traffic filter list */
    struct tcf_proto    *filter_list;
    int                 filtercnt;

#ifdef CONFIG_NET_SCH_HPDDSCP
    /* DSCP table */
    struct hpd_class    *dscp_table[64];
#endif

    /* Classes */
    struct hpd_class    *first_class;
    struct hpd_class    *default_class;
    struct hpd_class    *class_table[TC_HPD_CLASSES];
    u8                  highest_class;
};
```

Figure 4.4: Discipline data structure

defaults to 32 and can only be changed by recompiling the queuing discipline. The array is indexed by class numbers, which makes it possible to have empty slots in between classes. Empty spaces make it harder to loop through the classes using the class array, but this is solved by having the classes know who their neighbours are. The mechanism for this is explained below where I describe the class data structure in more detail.

To speed up access to key classes, the discipline structure includes two additional class pointers; one for the first class, and one for the default class of the discipline. Also the index of the highest (last) class on the array is stored for fast reference.

Filter Work-Around

As mentioned above, filters do not work correctly in the 2.4.18 kernel. To work around this defect I have added a method to assign traffic to classes based on the DSCP field of the IP packet. The DSCP field can have 64 different values, thus my solution has a DSCP lookup table with one class pointer entry for each DSCP value. This solution makes it possible for each DSCP value to assign traffic to a certain class, and to have multiple DSCP values assigning traffic to a single class.

Class Data Structure

DBHPD class structures form a doubly-linked list. I found this to be a good choice from the experience I had when implementing the DBHPD algorithm

for ALTQ framework in FreeBSD [17].

A class structure, shown in figure 4.5, includes two class structure pointers, one for the previous and one for the next class in the list. The class data structure also includes a queuing discipline pointer referring to the next level queuing discipline. Other data fields included for management purposes are a reference counter for _get() and _put() routines, a filter counter, a structure for collecting statistics, class number, EWMA type, and a class identifier.

```
struct hpd_class {
    /* Structural */
    struct hpd_class    *next_class;    /* forward and back pointers */
    struct hpd_class    *prev_class;    /* for a linked list */
    struct Qdisc        *q;             /* next level qdisc */
    int                 refcnt;         /* reference counter */
    int                 filtercnt;      /* filter counter */
    u32                 classid;        /* class id */
    struct tc_stats     stats;          /* generic statistics */

    /* Parameters */
    u8  class_num;                      /* class number */
    u8  ewma_type;                      /* EWMA type */
    u32 diff_param;                     /* differentiation parameter */
    u32 weight;                         /* delay weighting */
    u32 gamma;                          /* traffic weighting */
    u32 cycle;                          /* restart cycle time */
    u32 p_thresh;                       /* restart threshold */
    u32 n_coef;                         /* error proportion coefficient */
    u32 delay_limit;                    /* hard delay limit */

    /* Variables */
    u64 avg_delay;                      /* average delay value */
    u64 ewmar_sum;                      /* arithmetic sum for EWMA-r */
    u32 p_samples;                      /* EWMA-r packet counter */
    psched_time_t idle_time;            /* EWMA-r idle time */

    /* Time value ring buffer */
    psched_time_t *time_queue;
    u32 tq_length;                      /* queue length */
    u32 tq_first;                       /* indexes */
    u32 tq_last;
};
```

Figure 4.5: Class data structure

DBHPD algorithm has a good number of configuration parameters. Each class includes a delay differentiation parameter and a delay weighting factor as shown in eq. 3.1 & 3.2. The three EWMA estimators all use the weighting factor γ from eq. 3.3. EWMA-r estimator uses also a restart cycle time and a restart threshold from eq. 3.5. EWMA-pe estimator needs an additional parameter for calculating the proportional error multiplier n from eq. 3.6. For delay bounding, a class based delay limit parameter is introduced.

The average delay calculations need to know the previous hybrid delay

values. For this we need a variable for average delay of the class. EWMA-r requires also to keep track for the restart packet counter, idle time and an arithmetic sum of delays used when restarting. This adds three more variables to the class structure.

DBHPD calculations are based on delays of the packets in the queues. To know the delays the arrival time of each packet needs to be stored somewhere. I decided to store the arrival times in a ring buffer following the example presented in previous implementations [22, 17]. The ring buffer is basically an array with one time structure element for each place in the classes queue. Two values are used for indexing the ring buffer; one for the first packet in the queue and one for the last packet in the queue. These indexes tell us where to look for a current head of the line packet's arrival time value and where to store the new end of the line packet's arrival time. Circularity is achieved by storing the queue length in another value and cycling the indexes back to zero when they grow larger than the queue length.

Statistics Structure

For making things easier for a programmer and to standardize the API, each queuing discipline uses the same basic structure for collecting class statistics. This structure includes fields for the number of passed and dropped packets and bytes, the current queue length, and current byte and packet flow rates. The API also includes a way to add a discipline dependent statistics structure to the statistics report. When using DBHPD this discipline dependent structure includes information on the class's average delay and the current hold time at the head of the class's queue.

Options Structures

Options structures are used to control the queuing discipline from userland and to send information of queuing disciplines parameters to userland. DBHPD discipline uses three different options structures. The main options structure is the class options structure. This structure includes a field for every configurable parameter used by the DBHPD algorithm. These are the same parameters as described in the class structure section above. Most of

the fields contain only a single attribute, but the information of the EWMA type and whether the class is a default class of the discipline or not are combined into an 8-bit binary flags field.

The other two options structures are used for DSCP filter work-around. One of the structures is used to send a new DSCP lookup table entry from userland to kernel while the other is used to send an image of the whole DSCP lookup table from kernel to a userland application.

4.4.2 DBHPD Algorithm

DBHPD algorithm makes the queuing decisions based on normalized delay times of class queues. To be able to calculate the delays, the algorithm uses information of packet enqueuing and dequeuing times. Packet enqueuing time is stored in the class structure's time buffer during the enqueue routine. To avoid excessive time queries and ensure fairness between classes, the dequeuing time is checked only once at the beginning of the dequeue routine.

When compared against the simple EWMA estimator, the EWMA-r and the EWMA-pe estimators both use an additional set of calculations. For EWMA-r, it is important to know for how long a class has been idle, and how many packets have been passed since the last idle period. When using EWMA-pe, additional operations are added by the calculating of a weighting factor based on the difference between current and average delays.

Both main parts of the implementation, enqueue and dequeue functions, are included in appendix A. A listing of changed kernel source files can be found in appendix B.

Enqueuing a Packet

The enqueuing process is illustrated in figure 4.6. The main purpose of an enqueuing routine is to add a packet to the correct queue on one of the discipline's classes. To do this the packet needs to be classified. The usual way to do this is to call a helper routine, which checks the packet against any filters connected to the discipline. With DBHPD discipline the helper routine checks the incoming packet's DSCP value against a DSCP lookup table. If a match is found, the packet is added to the corresponding class's

queue. If no match is found a default class is used instead. If there is no default class the packet is dropped.

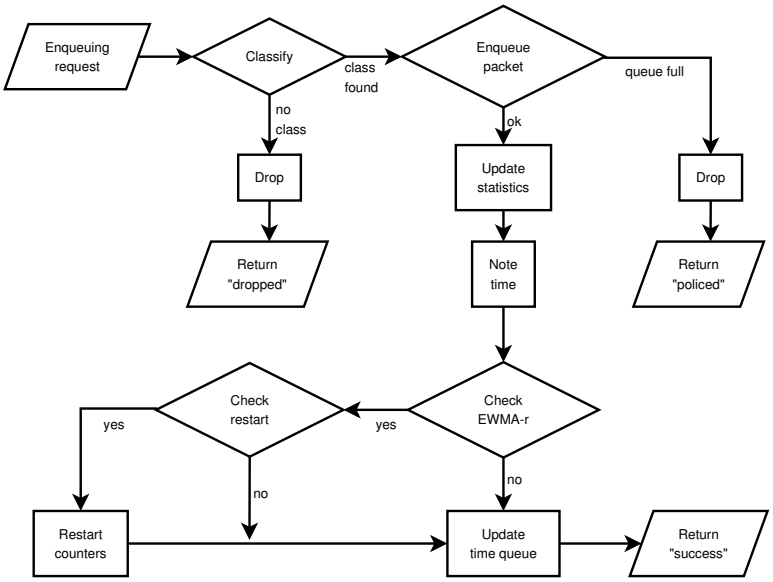

Figure 4.6: Enqueuing flowchart

After classifying the packet the enqueuing routine updates both the discipline's and class's statistics structures. The packet counters are incremented and the byte counters are increased by the byte count of the packet.

Any additional calculations a queuing algorithm requires are performed next. With DBHPD this means that the packet arrival time is noted and stored in the class's time value buffer.

When using simple EWMA or EWMA-pe, the enqueuing routine has now performed its task and returns. If using EWMA-r the restart conditions have to be checked. This is done by first checking if the queue was empty, i.e. if the size of the queue is one, because the packet has already been added to the queue. If the queue was empty before the arrival of the current packet then the time difference between the new packet and the last dequeued packet is calculated. If this time difference exceeds the defined idle time of the class, then restart is performed. Restarting zeroes the average delay, arithmetic sum and packet counter values.

Dequeuing a Packet

Dequeuing process is illustrated in figure 4.7. Dequeuing routine selects a class for dequeuing. This is usually done by going through all the classes of a discipline in a predefined order and calculating a comparison value for every class. The dequeuing decision is then based on this value. My implementation of the DBHPD discipline does this by calculating normalized weighted average queuing delay values starting with the highest numbered class and proceeding backwards along the linked list of classes. Normalized delay is stored in a 64-bit unsigned integer. The reason for this is the greater value range of 64-bit integers and the non-existence of negative delays.

The reason for backwards traversal lies within the condition clause I have chosen for checking which class has the highest normalized average delay. By checking whether or not the class's delay is greater than or equal to, instead of just greater than, the highest delay encountered this far allows initialization of the highest delay to zero without worrying about all the classes having a zero delay. By starting from the last class and progressing backwards allows us to give priority to the lower numbered class in the rare case of normalized delays being equal.

To be able to calculate the delay the dequeuing routine first takes a note of the current system time. The queuing times at the head of every class's queue are calculated against the same time value. Using the same time value for every class allows us to neglect the time spent by the dequeuing routine itself.

If a class has a set delay limit, and the queuing time is greater than this delay limit, a packet from the violating class is dequeued immediately. This is the only case where a higher numbered class can get a priority over an equal – i.e. a class which is also violating its delay limit – lower numbered class. This functionality also creates an interesting altough not practical application: if every class in a DBHPD using system has an unattainable delay limit, the system acts like it was using PRIQ instead.

If a class does not violate its delay limit a normalized average delay value is calculated and compared against other delay values calculated during this dequeuing cycle. The normalized average delay limit is calculated in a series

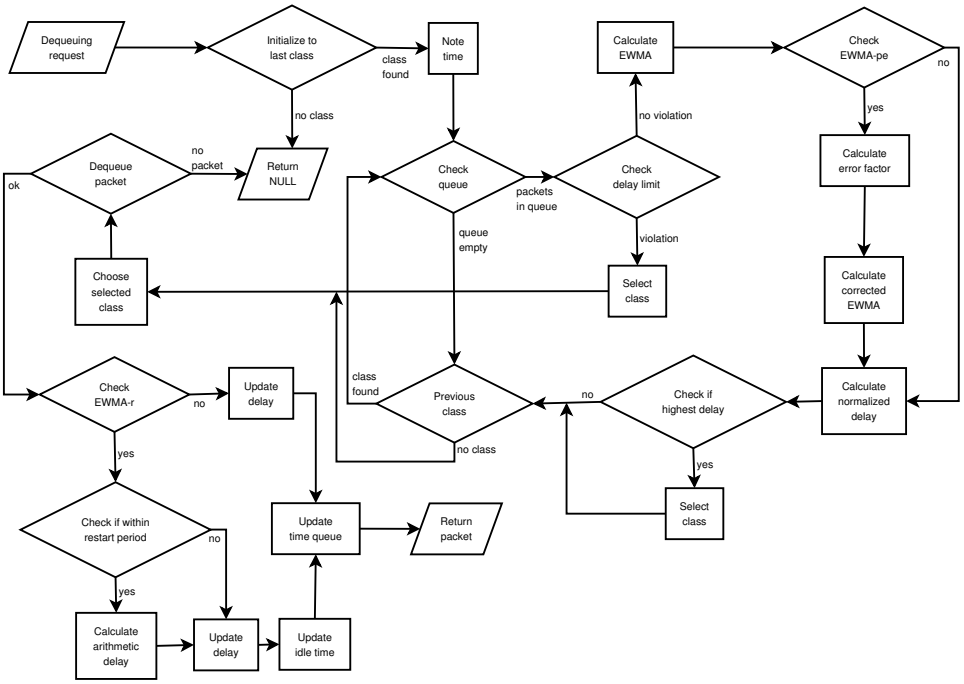

Figure 4.7: Dequeuing flowchart

of steps. The first step is calculating a new average delay value with:

```
class_delay = cl->gamma * hold_time +
              (1024 - cl->gamma) * cl->avg_delay;
```
(4.1)

Because the lack of floating point operations inside the Linux kernel, the class's `gamma` parameter is scaled between 0 and 1024 (2^{10}). The calculation in eq. 4.1 is performed regardless of the EWMA type. The non-normalized average delay is calculated regardless the class violating its delay limit or not.

If the class uses EWMA-pe for the delay calculations, the average delay is recalculated with the following code:

```
unsigned int n_thresh = cl->n_coef;
unsigned int n_gamma = n_thresh, i;
for (i = 0; i < n_thresh; i++) {
    if (hold_time > ((8 + n_gamma) * class_delay) >> 3)
        break;
    n_gamma--;
}
if (hold_time > class_delay)
    n_gamma = 1;
if (!n_gamma) {
    for (i = 0; i < n_thresh; i++) {
        n_gamma++;
        if (hold_time > ((8 - n_gamma) * class_delay) >> 3)
            break;
    }
}
class_delay = n_gamma * cl->gamma * hold_time +
              (1024 - n_gamma * cl->gamma) * cl->avg_delay;
```
(4.2)

This code searches the weighting factor n used in eq. 3.6. The method used for finding n includes two consecutive loops for finding the relative difference between current and average delays from eq. 3.7. The first loop starts from $1 + 0.125 * \texttt{n\_coef}$ and proceeds to 1 with steps of 0.125. The second loop starts from 0.875 and proceeds to $1 - 0.125 * \texttt{n\_coef}$ with steps of 0.125. The step of 0.125, or $\frac{1}{8}$, is chosen because division by eight can be quickly calculated with a bit shift operation.

After the average delay has been calculated, the normalized average delay is calculated with

```
norm_delay = cl->weight * class_delay +
             (1024 - cl->weight) * hold_time;
do_div(norm_delay, cl->diff_param);
```
(4.3)

where `weight` and `diff_param` are class dependent weighting and delay differentiation parameters. The function `do_div(...)` is a way to do arbitrary divisions inside the non-floating point enabled kernel environment.

If a class has greater than or equal normalized average delay than any of the classes calculated before it, the class gets selected and the class's number and average delay are saved in temporary variables. If the class uses EWMA-r, the current delay is saved as well.

After all the classes' delay values have been calculated and the class with the highest delay value has been found, a packet from that class gets dequeued. If the selected class uses EWMA-r and not enough packets to warrant using EWMA instead of arithmetic average have passed through the class's queue, the average delay value is recalculated as an arithmetic

average, and the beginning time of the idle period is also updated to the current time. The average delay value of the class is updated to the new value regardless of the EWMA type used.

4.4.3 Management and Statistics Collection

Most of the class management is done inside a single function. The function `hpd_change_class()` is used for both creating a new class and modifying an existing one as depicted in figure 4.8. This function's behaviour depends on whether the class given as a parameter exists or not. If the class exists, the class variables are updated to new values found in a class parameter structure given as a parameter.

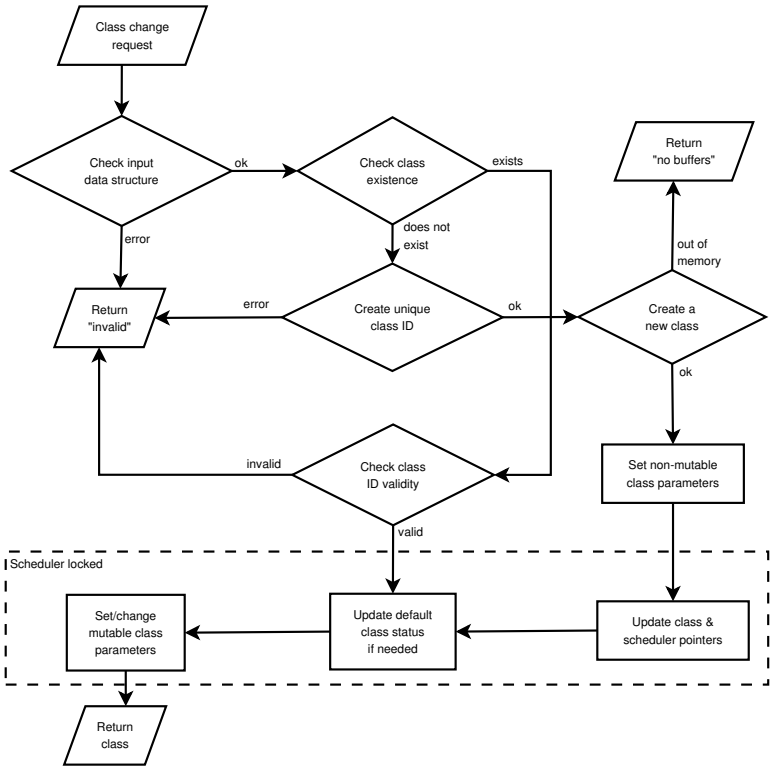

Figure 4.8: Class management flowchart

If the class does not exist a new class is created. First a class and parent identifiers given as a parameter to the main management function are checked against conflicts. If no conflicts exist, memory is allocated for

a new class structure and class variables for the new class are stored in it. The class is also linked to its appropriate place inside the discipline's class table. Finally the class is linked in its correct place within the linked list formed by all the classes connected to the discipline.

Class removal is performed by `hpd_delete()` function. A class can be removed if it is not the default class of the discipline and it has no filters attached to it. Upon removal the class is dereferenced from the linked list of classes and from the discipline's class table. The class's reference counter is decreased and if it reachers zero, the class is removed from the memory. If the counter does not reach zero the class is not removed at this time. The removal is done after the reference counter reaches zero by the function that releases the last reference. Default class can be removed by assigning another class as a default class of the discipline. The last class of the discipline can be removed only by removing the discipline itself.

Class and discipline statistics are updated only during an enqueuing operation. When a userland application requests statistics from the discipline the statistics structure is bundled with class and discipline parameters structures, and with an additional statistics structure. The bundled structures are then sent to the userland application via RTNetLink interface. The additional statistics structure includes a class's average and current delay values. It is up to the userland application to decide whether to display this information or not. The statistics reporting process is illustrated in figure 4.9.

4.5 User software

User application for controlling the packet queuing disciplines from outside the kernel is called *tc*. *Tc*, shortened from traffic control, has a modular structure, which makes it relatively easy to extend. To add a support for a new queuing discipline, a new control module needs to be written. The control module is automatically linked to the main application during build time.

A control module is required to implement the TC API described in section 4.3.3. The routines in the control module are used to extract param-

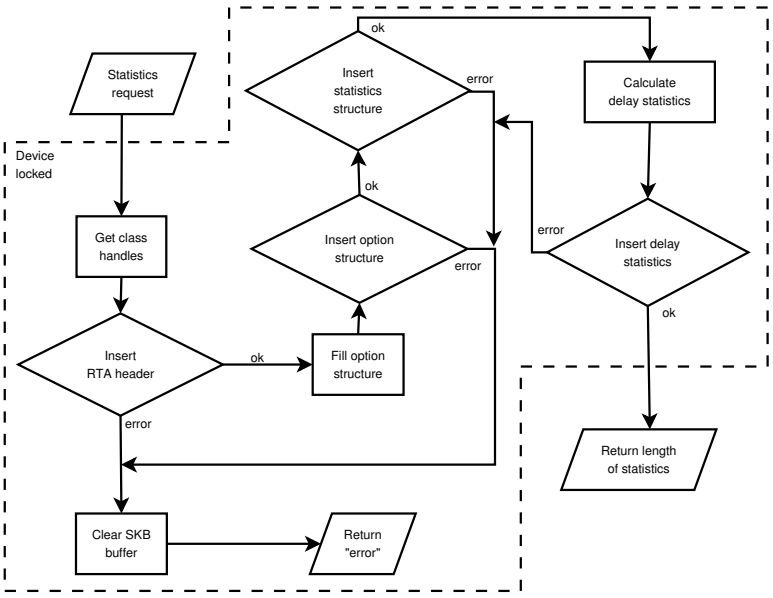

Figure 4.9: Statistics reporting flowchart

eters from command line into a data structure which can then be sent to kernel, and to display statistics information received from the kernel to the user. All communication between *tc* and the kernel facilities is performed using the RTNetLink interface.

Command line parameter parsing is done with helper functions found in *tc* library. These functions perform string matching and conversion to numerical form. The format of the parameter data structure is described in the Linux system header files and is identical to the format used inside the kernel. Parameters used with the DBHPD discipline are listed in table 4.1.

When *tc* queries the kernel for discipline or class statistics, the kernel returns the statistics in a bundle of structures. These structures include class and discipline parameters, basic class statistics like passed and dropped packets and bytes, and possible extra statistics. For the DBHPD discipline the extra statistics included are class's current and average delay.

The format in which the parameters and statistics are displayed is defined by individual control modules for queuing disciplines. Because the DBHPD discipline does not take any additional parameters (except when the filter work-around is enabled) only class parameters are shown. The

Table 4.1: Parameters and their descriptions

Discipline parameters		
dscp	n	: DSCP for directing traffic to a class
class	n	: Class number for traffic to be directed at
		:
Class parameters		
num	n	: Class's number
default		: Marks the class as a default one for the discipline
ewma	*xxx*	: EWMA type used
limit	n	: Hard delay limit for the class in microseconds
qlen	n	: Class queue length
ddp	n	: Delay differentation parameter (σ_i in eq. 3.2)
gamma	n	: Weighting factor between current and average delays (γ in eq. 3.3)
weight	n	: Weighting factor between short and long term delays (g in eq. 3.1)
restart	n	: Restart threshold when using EWMA-r ($p_{thresh,i}$ in eq. 3.5)
cycle	n	: Cycle time used when using EWMA-r ($cycle_i$ in eq. 3.5)
properr	n	: Number of 12,5% steps used with calculating correction factor n from eq. 3.6 with EWMA-pe as per eq. 4.2

class parameters are formatted identically with the command line used for creating the class. This is to avoid confusion and to make it easier to copy and compare settings. An example of HPD statistics display is given in figure 4.10.

The DSCP mapping used by the filter work-around described in section 4.4.1 is displayed in a tabular format. Each of the 64 DSCP values are shown with their corresponding class numbers. If no class number is assigned for the DSCP value, the class number defaults to the discipline's default class's number.

```
card3> tc -s class show dev eth3
class hpd 8001:8001 root leaf 8002: num 1 ewma ewma limit 0 qlen 50 ddp 1 gamma
512 weight 512 restart 25 cycle 2000 properr 8

 Sent 787720036 bytes 3636691 pkts (dropped 0, overlimits 0)
  hold time:     120
  average delay: 234435

class hpd 8001:8002 root leaf 8003: num 2 ewma ewma limit 0 qlen 50 ddp 4 gamma
256 weight 768 restart 25 cycle 2000 properr 8

 Sent 112533088 bytes 519528 pkts (dropped 0, overlimits 0)
  hold time:     13
  average delay: 680045

class hpd 8001:8003 root leaf 8004: num 3 ewma ewma limit 0 qlen 50 ddp 16 gamma
 768 weight 256 restart 25 cycle 2000 properr 8

 Sent 112525482 bytes 519521 pkts (dropped 0, overlimits 0)
  hold time:     68
  average delay: 1303871
```

Figure 4.10: Example of a statistics display

Chapter 5

Measurements

To validate the implemented packet scheduling algorithm we need to measure its performance. By careful measurements we can compare the implemented packet scheduler against existing alternatives. Based on these measurements and comparisons we can form an image of the performance and applicability of the new scheduler.

Measurements I have used can be divided into two categories: performance measurements and traffic simulations. Performance measurements are used to find out the theoretical limitations of the hardware and software. Traffic simulations are used to estimate the schedulers performance under real life traffic scenarios.

5.1 Measuring Performance

When we think about a routers performance, we have two factors to consider. The first factor is throughput, i.e. how much traffic the router can process. The second factor is delay, i.e. how long it takes for the router to process traffic.

I use two different ways of expressing throughput in my measurements. The first one is bits per second or *bps*. A *bps* value describes the amount of information a router can process. This amount depends on the router's buses' data transfer capability as well as the transmission rates of the network interfaces and the network itself.

The second way of expressing throughput is in terms of packets per second or *pps*. This method gives a more meaningful number for comparing routers' and scheduling algorithms' performance. This is because the amount of packets per second a router can process directly limits the *bps* value as well. For example, lets say that a router has a throughput value of $20kpps$. Now if the average size of a packet is $100B$, the throughput value expressed in bits per second is $20kpps * 100B * 8b/B = 16Mbps$. However, if the average packet size is $500B$, the previous calculation gives us a throughput of $80Mbps$.

The real numbers are not this linear. Usually a router can pass a larger number of smaller packets than larger ones. However, the throughput in terms of *bps* is still better with larger packets than smaller ones. Because of this non-linearity, measurements need to be performed with various packet sizes.

The number of packets per second a router can process is limited by the routers architecture and operating system. With NMS using Linux each arriving packet causes an interrupt and the number of interrupts the NMS CPU can handle is limited. This means that especially with smaller packets the actual throughput expressed in *bps* is a lot lower than the theoretical maximum throughput of the NMS networking interfaces ($100Mbps$). The results of the measurements presented in the following chapter tell us exactly how much these two values differ to further illustrate this point.

Delay is a measurement of the service time of a packet. A delay is a sum of two factors: transmission delay and queuing delay. Transmission delay is the time it takes for a packet to propagate through the network. This depends of the physical qualities of and encoding methods used in the network. Queuing delay is the time it takes for the networking equipment to process a packet. This is the sum of the time a packet spends waiting for service in different queues and the time a networking equipment needs to serve the packet. In a static, controlled network the transmission delay can be considered constant. This means that delay comparisons reflect only changes in the queuing delay, which is a good thing, as the queuing delay is the part which we are interested in when we are measuring the performance of a router.

5.2 Simulating Traffic

Performance measurements are fine as they are, but they can only give us theoretical values which may or may not have anything to do with the router's performance in a functioning network. Traffic simulations are used to get an approximation of how a router might function in a live environment. The idea of a traffic simulation is to expose a router to a realistic traffic mixture such as the router might experience when connected to a corporate or ISP's network.

To perform a traffic simulation an estimate of the traffic mix the device is likely to encounter is required. A likely scenario for packet scheduling is in networks where there exists types of traffic that need to be prioritized over other traffic types. Real-time traffic, such as VoIP or video, fits this description well. Other traffic types likely encountered in such an environment include short flows with random sized packets, most of which are small themselves, and long flows with packet sizes close to network's MTU. Examples of the first kind are HTTP requests (small packet) and responses (a flow of a few long packets ending with a small one). Examples of long flows are FTP transfers of large files.

Correct proportions of these traffic types are difficult to estimate. According to measurements made at MCI Telecommunications Corporation by Claffy et al. during 1997 and Thompson et al. during 1998, about 90 percent of Internet traffic uses TCP. Same measurements show that WWW traffic makes up roughly 75 percent of the total TCP traffic, making WWW the dominant application [25, 6]. Later measurements, like the ones collected in TKK Networking Laboratory and used by Ilvesmäki and Luoma, confirm the 90/10 TCP/UDP ratio but show a decrease in the percentage of WWW traffic [12]. This can be explained as a risen usage of P2P applications.

Based on these measurement results I have prepared three different traffic simulation scenarios. These scenarios are described in further detail at the end of this chapter.

5.3 Test Network

The test network comprises of six different LANs interconnected with five NMS routers. The topology of the network is illustrated in figure 5.1. Each router has a dedicated control interface, one interface for each LAN it functions as a gateway for, and one interface for each adjacent router it is connected to. The LANs are implemented within a single centralized switch which has been certified to be able to handle all the traffic necessary for the measurements without any problems. Links between routers are implemented as VLANs in a separate switch. This allows controlling and monitoring of the link speeds and states remotely.

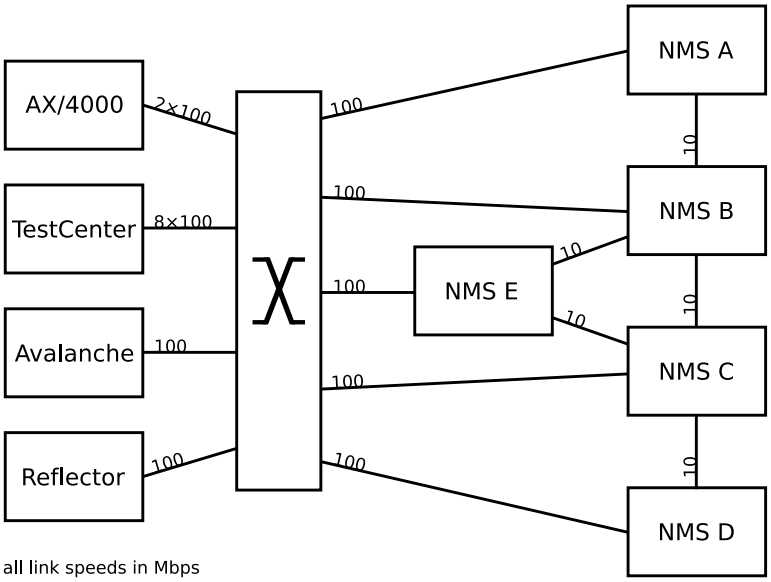

Figure 5.1: Topology of the test network

5.3.1 Measurement Devices

I have used three different measurement devices to run my tests. All of these devices are manufactured by Spirent Communications plc. First of these devices is Spirent TestCenter, which is a multi-purpose device designed for testing performance of network routers and switches. The second device is AX/4000 designed for network performance and quality of service testing.

The third device, or rather pair of devices, are Avalanche and Reflector. Avalanche is designed to simulate a large group of client computers while Reflector is designed to act as a simple server able to cope with a large number of clients. When combined these two devices can be used to test the load bearing capabilities of network and routing appliances.

All of these devices are connected to the same switch with the NMS LAN interfeces. Thus the measurement devices can easily be connected to any or all of the LAN interfaces by using VLANs

5.4 Test Scenarios

Two different measurement categories result in two different types of test scenarios. The first type of scenarios I call raw performance benchmarking. These are used to measure processing delays and theoretical maximum throughput of the routers. The second type of scenarios, traffic simulations, are used to compare the packet scheduling algorithms performance under real-like traffic. Chapters 6 and 7 are dedicated for presenting the results of these measurements.

5.4.1 Raw Performance Measurements

I use raw performance benchmarking to find out both the throughput and delays of the NMS hardware running different packet scheduling algorithms. The throughput tests are performed with a single router configuration while the delay tests use a chain of four routers. The tests are designed to reveal the limitations of the used hardware and the relative complexity between the packet scheduling algorithms. Thus the performance of the algorithms in the context of scheduling effectiveness is not validated.

Throughput tests

Throughput tests concentrate on throughput in terms of *pps* while using small packets. The packet sizes selected for these tests are 80, 160, 240 and 320 bytes. The packet sizes are limited to small packets only because of an unresolved bug in the network drivers, which causes the NMS kernel to

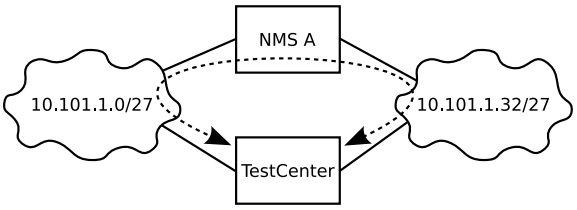

Figure 5.2: Single router configuration

panic when traffic rates rise over $90Mbps$.

The tests are designed to yield information about the routers throughput in terms of packets per second, which tends to be the limiting performance factor when the median of packet sizes is low. According to statistics from NASA AIX measured during 1999 to 2000, the low median size of packets fits the traffic profile of the Internet [31].

To see the effects of a packet scheduling algorithm to the throughput, the test run is repeated with FIFO, PRIQ, CBQ and HPD schedulers. With FIFO scheduler only one traffic class is used. With classful PRIQ, CBQ, and HPD schedulers the test is performed with both one and three traffic classes to reveal the effect of having multiple classes on the performance. The classes are identified by their DSCP values.

Delay tests

Delay tests are designed to show how long it takes for the router to process a single packet. I use a constant load of the total of $2,5Mbps$ for all the delay tests. The test is performed with 80, 160, 320, 640 and 1280 byte packet sizes. Like the throughput tests, delay tests are also performed with four different packet scheduling algorithms, FIFO, PRIQ, CBQ and HPD, and with one and three traffic classes per algorithm. The exception to the latter is FIFO scheduler with which only a single traffic class is used.

Delay tests are made with four router configuration. The flow of packets is directed from the end of the router chain towards the beginning of the chain. More traffic is added on each router, thus the outgoing interface of every router has more packets to schedule than the previous one in the chain. The additional packets cause the ordering of the already scheduled

incoming packets to be disrupted, thus requiring each router to perform more scheduling calculations.

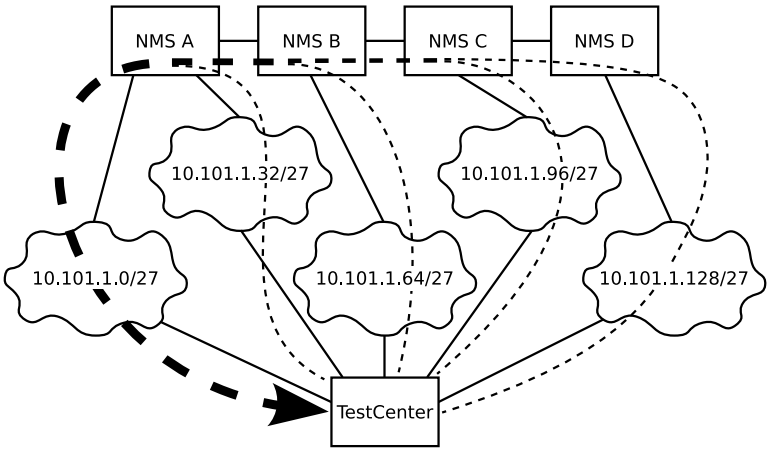

Figure 5.3: Four router configuration

Delay statistics are collected by flow, identified by source address and DSCP value. Single test run results in delay statistics over one to four hops. These results give a rough estimate of the time differences required to perform scheduling calculations between different scheduling algorithms.

5.4.2 Traffic Simulations

I use traffic simulations to analyze the performance of the packet scheduling algorithms under real-like traffic conditions. These simulations give an estimate of the effect to network traffic quality caused by using a packet scheduling algorithm. This effect can be measured by creating a real-like traffic mix and analyzing the characteristics of the traffic flow before and after scheduling.

I have decided to use four different traffic classes in my traffic simulations. These classes are HTTP, FTP, VoIP and noise. HTTP and FTP classes represent non-real-time traffic and are considered to be tolerant of delay variations. The traffic in these classes is pure TCP. Difference between HTTP and FTP classes is that the FTP class includes mostly MTU sized packets (long files) while the HTTP class has more varied packet sizes (lots of requests for small files).

Table 5.1: Traffic mixes used in simulations [%]

	Mix 1		Mix 2		Mix 3	
	TCP	UDP	TCP	UDP	TCP	UDP
HTTP	70	0	22	0	22	0
FTP	10	0	5	0	5	0
VoIP	0	8	0	4	3	1
Noise	10	2	60	9	60	9

VoIP class represents delay variation intolerant real-time traffic which the scheduling should prioritize. The traffic in this class can be either UDP or TCP depending on scenario. Connectionless UDP traffic would be the better option from network point of view, but because some ISPs and firewall solutions handle TCP better than UDP, some VoIP applications, like Skype, are using TCP instead. To account for this, I use scenarios with either only UDP or a mixture of TCP and UDP traffic. The exact mixtures are shown in table 5.1.

Noise class is used for traffic that can not be classified and thus can be regarded as a background noise from the scheduling algorithm's point of view. Traffic falling into this class is not prioritized. Noise can include e.g. DNS requests and P2P traffic, and has both TCP and UDP components. Based on recent, unpublished measurements performed in TKK Networking Laboratory, most of the traffic in the Internet falls into noise category. To reflect this, I have devised scenarios having both small and large amounts of noise.

Traffic mixes are presented in table 5.1. Mix 1 represents a low-noise environment where all the real-time traffic uses UDP. Mix 2 also has all the real-time traffic using UDP, but this time the noise level is considerably higher. In mix 3 most of the real-time traffic uses TCP while the noise level stays high.

Different classes are identified by DSCP field and not by e.g. source and destination port pairs. This allows for better control and easier analysis but also causes the classifier to be more exact than in real life situations, where noise appears in every class.

Chapter 6

Raw Performance Measurements

6.1 Throughput Measurements

6.1.1 Test Setup

The purpose of these tests are not to validate the scheduling algorithm but to validate its implementation. To be successfully implemented, the discipline should not cause any more strain to the system than any of the already existing disciplines. The network layout is shown in the figure 5.2. The configuration parameters used for the disciplines are listed in table 6.1.

The aim of the configuration is not to simulate any real scenario but only to cause the system to perform an amount of instructions comparable to a real situation.

6.1.2 Results

Test results displayed in figure 6.1 show that no single scheduling discipline outperforms others. The differences between disciplines are very small with every measured packet size and none of the disciplines shows best or worst result at every measurement point. This means that the complexity of the scheduling discipline has no effect on theoretical maximum throughput with NMS hardware.

Table 6.1: Discipline parameters for raw performance measurements

FIFO	
No additional settings	
PRIQ	
class 1	priority 0
class 2	priority 1
class 3	priority 2, default
CBQ	
root class	bandwidth 100 Mbps
	average packet size 256 B
class 1	priority 1, rate 80 Mbps
	per round allotment 300 B
	average packet size 200 B
class 2	priority 2, rate 10 Mbps
	per round allotment 300 B
	average packet size 200 B
class 3	priority 3, rate 10 Mbps
	per round allotment 300 B
	average packet size 200 B
HPD	
class 1	delay differentiation parameter 1
	gamma 512, weight 512
class 2	delay differentiation parameter 4
	gamma 256, weight 768
class 3	delay differentiation parameter 16
	gamma 768, weight 256

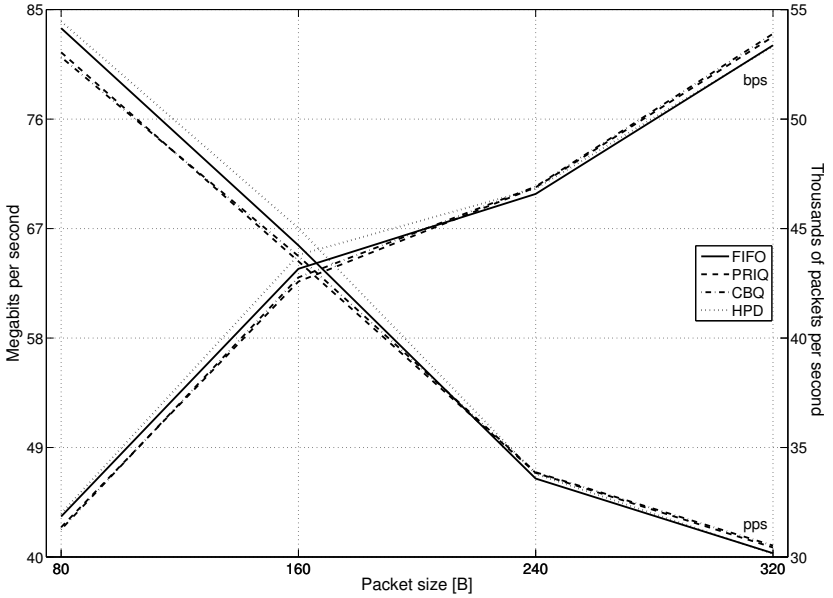

Figure 6.1: Throughput results summary

All the disciplines are able to achieve minimun throughput of over $40Mbps$ and $25kpps$. They are able to saturate a $10Mbps$ link like the one I am using with traffic simulations in the next chapter. Also the *bps* vs. *pps* profile is behaving as expected; when packet size grows the number of packets per second the NMS is able to process goes down but the amount of data going through rises.

6.2 Delay Measurements

6.2.1 Test Setup

The network layout for delay measurements is shown in figure 5.3. Each source sends data with $2,5Mbps$ in bandwidth, totaling to $10Mbps$ maximum bandwidth at the final router. This is well below the smallest maximum throughput the throughput test results show, thus the limitations in hardware should not affect the resulting latencies.

The discipline configurations are identical to the ones shown in table 6.1

except for the following changes:

- CBQ configured rates dropped from 80/10/10 Mbps to 8/1/1 Mbps
- class order rotated at each router:
  - NMS A: 1 / 2 / 3
  - NMS B: 2 / 3 / 1
  - NMS C: 3 / 1 / 2
  - NMS D: 1 / 2 / 3

These changes were made to ensure even workloads with all classes in every router.

6.2.2 Results

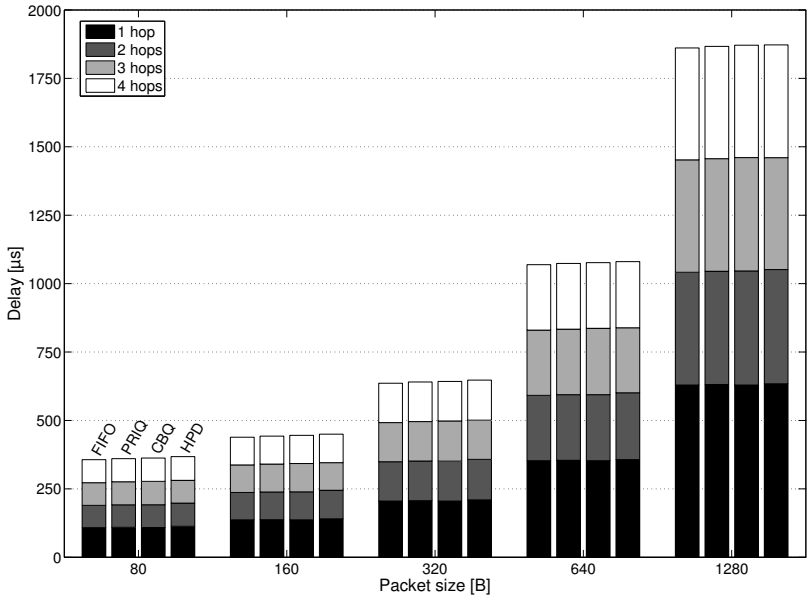

Figure 6.2: Delay results summary

The results displayed in figure 6.2 show that delay differences between different scheduling disciplines are very small. Taken that the calculations

used in disciplines' differ in amount and complexity, this most likely means, that the majority of the delays are caused by either hardware or other parts of software than packet scheduling.

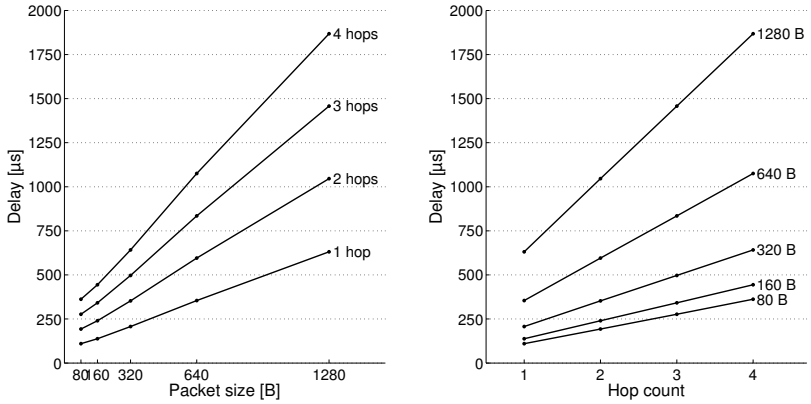

Figure 6.3: HPD delays per packet size and hop count

When delays are plotted against packet sizes or number of hops as in figure 6.3, the results fit on a single line. This was to be expected as every additional byte and every additional hop translates to a constant increase in transfer delays.

6.3 Result Summary

Neither one of the raw performance tests shows any difference between the scheduling disciplines with the NMS hardware. This is actually a good thing considering that in light of these results the traffic simulations will show algorithmical differences and not differences caused by the effectiveness of discipline implementations.

The hardware can forward over $40Mbps$ of traffic with each measured packet size. This means that the NMS is easily able to saturate a $10Mbps$ link, like the one I use in traffic simulations. Thus neither the implementation of the discipline or the limitations of the hardware will play any significant role in the traffic simulation measurements. Any difference they will show is based purely on algorithmical differences, as it should be.

Chapter 7

Traffic Simulations

7.1 Test Setup

Traffic simulations should reflect real life traffic scenarios. This means that the scheduler configurations should be close to real environments as well.

In these simulations the traffic in the network is divided into four different classes: VoIP, WWW, FTP and noise. Of these VoIP is given the highest priority - users making voice calls are most susceptible to low network quality after all. WWW and FTP are prioritized the same, the difference being in traffic allotments with CBQ and rate of adaptivity with HPD.

7.1.1 Scheduler Configuration

With priority queuing the classes are prioritized as follows: highest priority is given to VoIP. Next in priority order is WWW and FTP. Noise has the lowest priority.

With CBQ the root class is first divided to two parts. An isolated share large enough to cover all VoIP traffic is allocated for VoIP. The rest of the available bandwidth is shared between WWW, FTP and noise. WWW and FTP both get a share that is a close match to their bandwidth usage and are allowed to borrow from each other and from the noise class. Noise gets a very small share but is allowed to borrow bandwidth from FTP and WWW if available. For borrowing calculation purposes, the classes are prioritized in order: WWW, FTP, noise.

HPD is configured to use EWMA-R with VoIP class and regular EWMA with other traffic classes. VoIP class has a hard delay limit of 2 ms. VoIP and WWW classes use more aggressive delay adaptation than FTP and noise classes. The delay differentation parameters for the classes are 1 for VoIP, 2 for WWW, 4 for FTP, and 16 for noise.

Configuration parameters for each discipline are shown in table 7.1.

7.1.2 Traffic Mixes

The traffic is generated with Spirent Avalanche and Reflector measurement devices. These devices can simulate client-server environment with UDP and stateful TCP connections. Of the traffic classes WWW and FTP are self-explainatory. VoIP is simulated with both TCP and UDP SIP connections. UDP noise is made up with DNS requests and TCP noise consists of short telnet connections.

7.2 Results

7.2.1 Connection Analysis

With a low-noise traffic mixture the classful scheduling disciplines have a bit higher packet loss percentage than the classless FIFO. However the number of succesful connections is almost equal across the board. All the unsuccesful connections are either in high traffic WWW class or in the low priority noise class. There are no transaction failures in either VoIP or FTP class, which both have lower traffic and longer connections than the other two classes.

When a higher percentage of noise is introduced to the system, the difference in successes between high and low priority classes starts to show. There are still no interrupted transactions with VoIP and FTP classes. WWW success percentage also remains close to 100 % while the success percentage in the low priority noise class drops under 90 %. With classless FIFO discipline both the WWW and noise success rates drop close to 90 % and the packet loss is greater than with the other disciplines.

The differences between UDP and TCP using VoIP traffic in traffic mixes 2 and 3 are miniscule. This is not surprising taken that the NMS router

Table 7.1: Discipline parameters for traffic simulations

FIFO	
No additional settings	
PRIQ	
VoIP	priority 0
WWW	priority 1
FTP	priority 1
Noise	priority 2, default
CBQ	
Root	bandwidth 10 Mbps
	average packet size 256 B
VoIP	priority 1, rate 1 Mbps, isolated
	per round allotment 256 B
	average packet size 200 B
Other	priority 2, rate 9 Mbps
	per round allotment 256 B
	average packet size 200 B
Other - WWW	priority 3, rate 7.3 Mbps
	per round allotment 256 B
	average packet size 200 B
Other - FTP	priority 4, rate 1.5 kbps
	per round allotment 256 B
	average packet size 200 B
Other - Noise	priority 5, rate 200 kbps
	per round allotment 256 B
	average packet size 200 B
HPD	
VoIP	delay differentiation parameter 1
	EWMA with restart, delay limit 2 ms
	gamma 896, weight 128
WWW	delay differentiation parameter 2
	gamma 768, weight 256
FTP	delay differentiation parameter 4
	gamma 256, weight 768
Noise	delay differentiation parameter 16
	gamma 128, weight 896, default

Table 7.2: Connection statistics summary for a low noise traffic mix

	FIFO	**PRIQ**	**CBQ**	**HPD**
Packet loss [%]	5.03	5.81	5.51	5.65
– Client to server	6.90	8.30	7.88	8.27
– Server to client	3.42	3.63	3.45	3.35
Succesful transactions [%]				
– WWW	99.77	99.81	99.73	99.78
– FTP	100.0	100.0	100.0	100.0
– VoIP	100.0	100.0	100.0	100.0
– Noise	99.51	99.72	99.47	99.45

Table 7.3: Connection statistics summary for a high noise traffic mix

	FIFO	**PRIQ**	**CBQ**	**HPD**
Packet loss [%]	4.43	3.94	3.73	3.80
– Client to server	5.15	4.32	4.12	4.39
– Server to client	3.81	3.62	3.42	3.32
Succesful transactions [%]				
– WWW	88.46	99.76	99.66	99.64
– FTP	100.0	100.0	100.0	100.0
– VoIP	100.0	100.0	100.0	100.0
– Noise	90.13	88.62	88.63	86.73

performing classifying and scheduling is stateless and blind to the IP sub-protocol used.

A summary of the major differences in connection statistics between the tested scheduling disciplines are presented in table 7.2 for a low noise traffic mix and in table 7.3 for a high noise traffic mix. The succesful transactions in the statistics show the percentage of answered service requests the clients have made.

7.2.2 Delay Analysis

With a low noise traffic mix and 5 % packet loss the delay profiles of the different scheduling disciplines are close to identical. CBQ seems to achieve a bit lower delays than the other disciplines. FIFO and PRIQ are close seconds while HPD's delay profile suffers from quite a heavy tail.

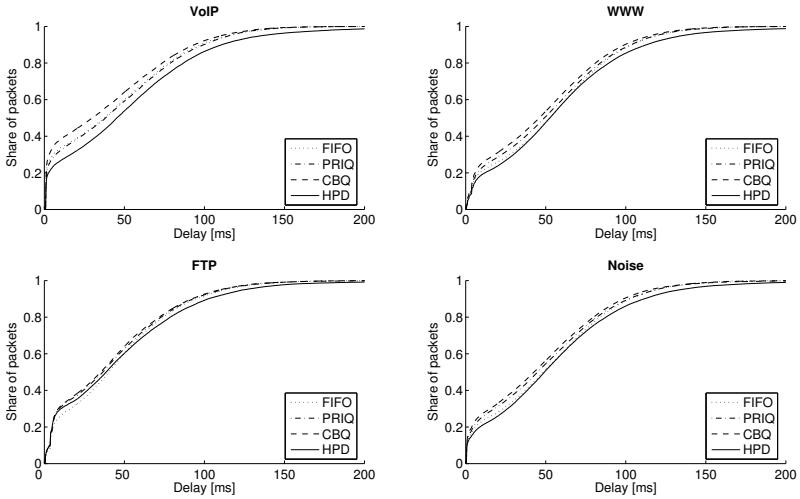

Figure 7.1: Cumulative delay distribution with low noise traffic mix

Differences between traffic classes are not so obvious. The cumulative distribution graphs in figure 7.1 show lower delays with higher priority classes, but when comparing WWW and Noise classes against VoIP and FTP classes one can also come up with a conclusion that lower traffic classes have more lower delay packets than higher traffic classes. The heavy tail of the HPD discipline can be seen with every class in figure 7.1 but is more easily noticeable in figure 7.2 which shows VoIP class delay distributions with HPD and CBQ disciplines.

When more noise is added to the traffic mix the situation changes a bit. If packet loss is kept close to 5 %, the amount of traffic needs to be reduced. In my tests the low noise traffic mix achieved $7Mbps$ throughput with 5 % loss where the higher noise mixes could only go up to $4Mbps$ without the packet loss growing over 5 %. At these rates the delay profiles of the different disciplines did not show significant differences between each other, as can be seen from figure 7.3. The tailing with HPD discipline is still visible. The shorter delay times in total can be explained by the lower workload of the system as the total throughput in terms of *bps* is lower.

If the throughput is kept at $7Mbps$ the total number of packets increases and more differences between the scheduling disciplines start to appear, as

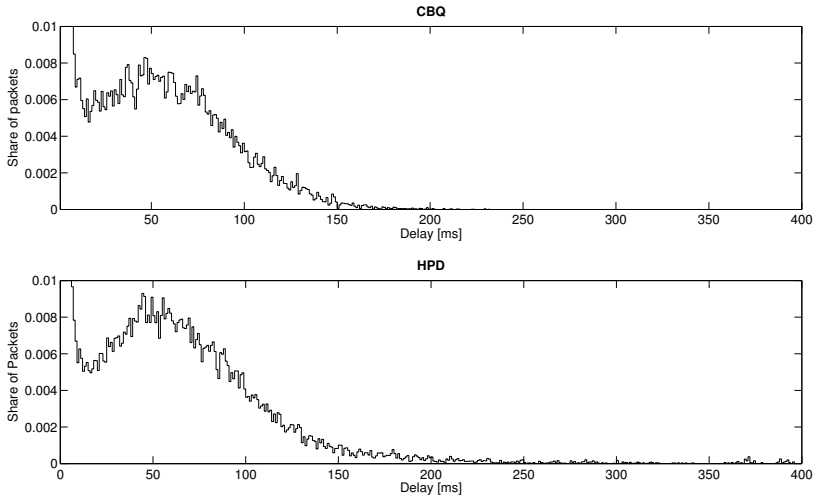

Figure 7.2: VoIP class delay distribution with low noise traffic mix

can be seen from figure 7.4. The packet loss increases to 35 % as the system is under chronic overload. FIFO, using no extra calculations, continues to function as before, but PRIQ and CBQ both start to lag behind.

The effect on HPD is more drastic: the tail grows and the distribution becomes almost linear. This behaviour did not show up with raw performance testing in the previous chapter, which showed little or no difference between CBQ and HPD delays. Also the number of succesful connections between different disciplines is not affected. This behaviour would require further research, but taken the problems with the outdated NMS environment should be done with more modern equipment.

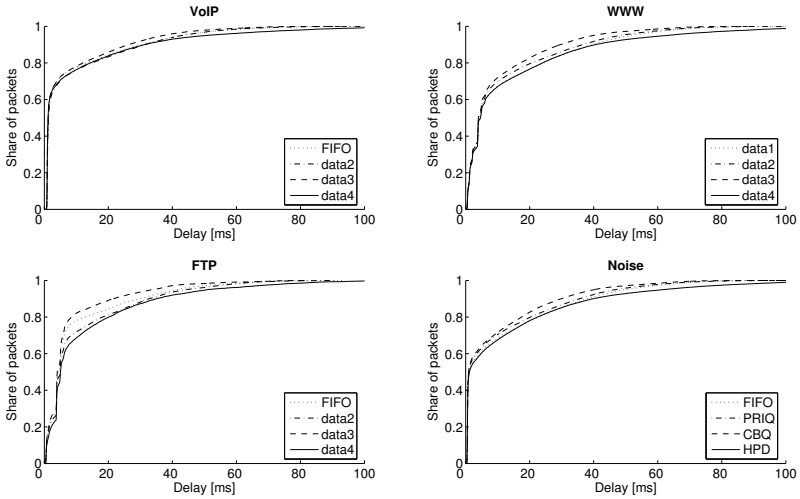

Figure 7.3: Cumulative delay distribution with high noise traffic mix

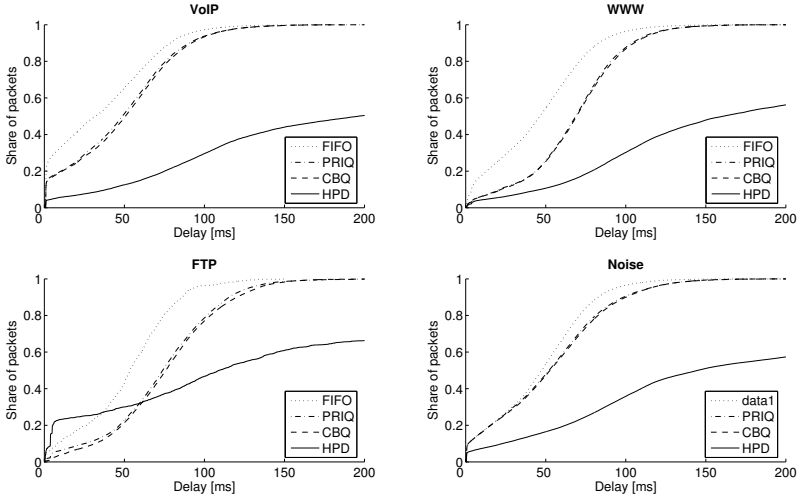

Figure 7.4: Cumulative delay distribution with chronic overload

Chapter 8

Conclusions

8.1 About Linux

Linux as an operating system is a good platform for embedded network appliances. The hardware support and availability of different software packages is excellent. The open nature of the source code makes it possible for developers to extend and modify already existing software with more ease and freedom than with proprietary software packages.

Because of the open nature, the quality of available programs is not always of the highest grade. There is less monetary incentive for manufacturers of external software to ensure that their programs work. However, the availability of the source code makes it possible to perform quality assurance for any planned software taken that there is enough resources available for such a task.

The version of Linux kernel used in this work is already outdated and I would not recommend it as a base for any new projects. The Linux networking API was rewritten between versions 2.4 and 2.6 and the new API does outperform the old in terms of performance and scalability [30].

8.2 About NMS Hardware

NMS hardware is based on a good idea but is terribly outdated. The original developer of the hardware has since gone bankrupt and the product was never finished. This shows mostly in driver problems; the FSR bus

driver causes the kernel to panic when the load rises over $90Mbps$ and the networking driver only works reliably in $100Mbps$ or $10Mbps$ half-duplex mode.

The hardware is compatible with Linux kernel version 2.4 only, which was the stable kernel at the time of the hardware's original development. As the 2.6 kernel has better networking performance than the 2.4 one, there is little else than purely academic reasons to continue development with the current NMS platform.

The networking requirements have also grown since the original release of the hardware. Both copper and fiber optic links of $1Gbps$ speeds and more have become reality. As such there is next to none demand for routers capable of only $100Mbps$ speeds without any possibility for upgrade.

8.3 About HPD Algorithm

HPD algorithm is an interesting one. The ability to prioritize traffic based on observed delays is important when assuring networking quality for delay intolerant traffic types. The current implementation can be reduced to work like simple priority queuing or become as versatile as CBQ. The calculations involved equal CBQ in terms of system load as can be seen from raw performance measurement delay statistics. The raw throughput is on par with other scheduling algorithm implementations. In previous measurements DBHPD with basic EWMA estimator performs on par or better than CBQ when delays and link utilization are considered [22].

The traffic simulation results reveal an alarmingly long tail with HPD delay distribution. This tail far exceeds the tail of the implementations of the other alogirhtms tested. The reason behind this tail remains unclear and warrants some future research. It is likely that it is caused by either the implementation or the hardware as it doesn't show up in analytical simulations on which my implementation is based on.

The hardware used in this work does not bring out any significant differences between the different scheduling algorithms. This is partly because the FSR bus driver does not allow running the hardware to its full capability in $100Mbps$ networks and the network driver does not allow using full-duplex

mode in $10Mbps$ networks, thus causing excessive delays. Another reason is the Linux traffic control architecture which places shaping before queuing in the packets path, which causes any artificially created bottleneck to be at the wrong side of the networking kernel.

8.4 Future Work

HPD algorithm warrants some more research and comparisons against other scheduling algorithms. Analytical work for this is already done, but simulations with actual hardware and realistic traffic are still lacking. The algorithm is currently implemented on FreeBSD 5.4 and Linux 2.4 kernels, which both are already outdated. For further research the algorithm should be ported to either or both of FreeBSD 6.0 and Linux 2.6 kernels and ran on more modern hardware.

The delay calculations involve large numbers as the timescale of the kernel is nanoseconds and a packet can spend more than a millisecond in a queue. For this reason the algorithm should be tested with 64 bit processors and kernel as well. As modern PCs are able to handle packet forwarding and routing up to gigabit speeds, and feature 64 bit processors, they would be ideal for rapid prototyping and testing of future versions of the HPD scheduler. Another suitable platform would be networking processor cards, which have become available from several manufacturers, and are able to run Linux and some other open source operating systems.

Bibliography

[1] Almesberger, W., Linux Network Traffic Control – Implementation Overview, Ecole Polytechnique Fédérale de Lausanne School of Computer & Communication Sciences, April 1999

[2] Antila, J. and Luoma, M., Robust Delay Estimation of an Adaptive Scheduling Algorithm, in Proceedings of QoS-IP 2005, February 2005

[3] Antila, J. and Luoma, M., Scheduling and Quality Differentiation in Differentiated Services, in Proceedings of MIPS 2003, November 2003

[4] Bailey, J., Building a cross-compiler, December 1999, (ref. 12.1.2007), [http://www.nongnu.org/thug/cross.html]

[5] Boxman, J., A Practical Guide to Linux Traffic Control, February 2005, (ref. 12.1.2007), [http://trekweb.com/~jasonb/articles/traffic_shaping/index.html]

[6] Claffy, K., Miller, G. and Thompson, K., The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone, in Proceedings of INET'98, July 1998

[7] Devera, M., Hierachical token bucket theory, May 2002, (ref. 12.1.2007), [http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm]

[8] Dovrolis, C., Stiliadis, D. and Ramanathan, P., Proportional Differentiated Services: Delay Differentiation and Packet Scheduling,

IEEE/ACM Transactions on Networking Vol.10 Issue 1, February 2002

[9] Floyd, S. and Jacobson, V., Link-sharing and Resource Management Models for Packet Networks, IEEE/ACM Transactions on Networking Vol.3 Issue 4, August 1995

[10] Freescale Semiconductor, MPC6260 datasheet, Freescale Semiconductor, Inc., November 2004

[11] Herrin, G., Linux IP Networking, University of New Hampshire Department of Computer Science, May 2000, (ref. 12.1.2007), [http://www.cs.unh.edu/cnrg/gherrin/linux-net.html]

[12] Ilvesmäki, M. and Luoma, M., Characterizing Internet flows with aggregated TCP/UDP source port measurements, in Proceedings of IPS-MOME 2006, February 2006

[13] IProute2 sources, developer.osdl.org, (ref. 12.1.2007), [http://developer.osdl.org/dev/iproute2/download/]

[14] Jain, A., Linux Networking Subsystem, Linux Network Documents, May 2002

[15] Kegel, D., Building and Testing gcc/glibc cross toolchains, December 2006, (ref. 12.1.2007), [http://kegel.com/crosstool/]

[16] Lamb, M., iproute2+tc notes, July 2006, (ref. 12.1.2007), [http://snafu.freedom.org/linux2.2/iproute-notes.html]

[17] Lamminen, O-P., An Adaptive Packet Scheduling Algorithm for the ALTQ framework in the FreeBSD 5.4 Kernel, Helsinki University of Technology Networking Laboratory, March 2006

[18] LinuxDevices.com, Snapshot of the Embedded Linux market – March, 2004, Ziff Davis Publishing Holdings Inc., March 2004, (ref. 12.1.2007), [http://www.linuxdevices.com/articles/AT8693703925.html]

[19] Linux kernel v2.4 sources, kernel.org, (ref. 12.1.2007), [http://www.kernel.org/pub/linux/kernel/v2.4/]

[20] Linux Networking Kernel, Linux Network Documents, February 2003

[21] Mosnier, A., Embedded/Real-Time Linux Survey, 4Real AB, July 2005

[22] Nieminen, J., Luoma, M. and Paju, A., Implementation and Performance Measurements of a delay-bounded HPD algorithm in an ALTQ-based router, in Proceedings of CoNext 2005, October 2005

[23] Raatikainen, P. and Zidbeck, J., Frame synchronized ring and its implementation, VTT Publications 217, 1995, ISBN 951-38-4655-5

[24] Stoica, I., Zhang, H. and Ng, E., A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services, in Proceedings of ACM SIGCOMM '97, September 1997

[25] Thompson, K., Miller, G. and Wilder, R., Wide-Area Internet Traffic Patterns and Characteristics, IEEE Network, November/December 1997

[26] Telegraph.co.uk, Connected Comment, Telegraph Media Group Ltd., April 2004, (ref. 12.1.2007), [http://www.telegraph.co.uk/connected/main.jhtml?xml=/connected/2004/02/04/ecnconc04.xml]

[27] User-Mode Linux Project, The User-mode Linux Kernel Home Page, The User-Mode Linux Project, (ref. 12.1.2007), [http://user-mode-linux.sourceforge.net/]

[28] Vlasenko, D., BusyBox: The Swiss Army Knife of Embedded Linux, OSL Open Source Lab, (ref. 12.1.2007), [http://www.busybox.net/about.html]

[29] Welte, H., The journey of a packet through the linux 2.4 network stack, September 2000, (ref. 12.1.2007), [http://www.sunbeam.ranken.de/projects/packetjourney/]

[30] von Leitner, F., Benchmarking BSD and Linux, November 2003, (ref. 24.5.2007), [http://bulk.fefe.de/scalability/]

[31] www.caida.org, Packet Length Distribution, Cooperative Association for Internet Data Analysis, (ref. 31.1.2007), [http://www.caida.org/analysis/AIX/plen_hist/]

Appendix A

Enqueue and Dequeue Functions

```
static int hpd_enqueue(struct sk_buff *skb, struct Qdisc *sch)
{
    struct hpd_class *cl = hpd_classify(skb, sch);
    int ret = NET_XMIT_POLICED;

    if (cl && cl->q) {
        if ((ret = cl->q->enqueue(skb, cl->q)) == 0) {

            sch->q.qlen++;

            sch->stats.packets++;
            sch->stats.bytes += skb->len;
            cl->stats.packets++;
            cl->stats.bytes += skb->len;

            PSCHED_GET_TIME(cl->time_queue[cl->tq_last]);

            if (cl->ewma_type == TCF_HPD_EWMAR) {
                if (cl->q->q.qlen == 1) {
                    if (PSCHED_TDIFF(cl->time_queue[cl->tq_last],
                                     cl->idle_time) > cl->cycle) {
                        cl->avg_delay = 0;
                        cl->ewmar_sum = 0;
                        cl->p_samples = 0;
                    }
                }
            }

            if (++cl->tq_last >= cl->tq_length)
```

```
                cl->tq_last = 0;

            return NET_XMIT_SUCCESS;
        }
        cl->stats.drops++;
    } else {
        kfree_skb(skb);
        ret = NET_XMIT_DROP;
    }

    sch->stats.drops++;
    return ret;
}

static struct sk_buff *hpd_dequeue(struct Qdisc *sch)
{
    struct sk_buff *skb = NULL;
    struct hpd_sched *q = (struct hpd_sched *)sch->data;
    struct hpd_class *cl = q->class_table[q->highest_class];
    struct hpd_class *selected_class = q->default_class;
    psched_time_t now;
    psched_tdiff_t hold_time, selected_hold_time = {0};
    u64 class_delay, norm_delay, max_delay = 0, selected_delay = 0;

    if (!cl || !sch->q.qlen)
        return NULL;

    PSCHED_GET_TIME(now);

    while (cl != NULL) {
        /* Check only non-empty classes. */
        if (cl->q && cl->q->q.qlen) {
            hold_time = PSCHED_TDIFF(now, cl->time_queue[cl->tq_first]);
            /* Immediately select class violating its delay limit. */
            if (cl->delay_limit > 0 && hold_time > cl->delay_limit) {
                selected_class = cl;
                selected_delay = cl->gamma * hold_time +
                                 (1024 - cl->gamma) * cl->avg_delay;
                goto class_selected;
            }

            /* Calculate EWMA result. */
            class_delay = cl->gamma * hold_time +
                          (1024 - cl->gamma) * cl->avg_delay;

            if (cl->ewma_type == TCF_HPD_EWMAPE) {
                unsigned int n_thresh = cl->n_coef;
                unsigned int n_gamma = n_thresh, i;
```

```
                for (i = 0; i < n_thresh; i++) {
                    if (hold_time < ((8 + n_thresh) * class_delay) >> 3)
                        break;
                    n_gamma--;
                }
                if (!n_gamma) {
                    for (i = 0; i < n_thresh; i++) {
                        n_gamma++;
                        if (hold_time < ((8 + n_thresh) * class_delay) >> 3)
                            break;
                    }
                }
                class_delay = n_gamma * cl->gamma * hold_time +
                              (1024 - n_gamma * cl->gamma) * cl->avg_delay;
            }

            norm_delay = cl->weight * class_delay +
                         (1024 - cl->weight) * hold_time;
            do_div(norm_delay, cl->diff_param);

            /* Select class with highest result. */
            if (norm_delay >= max_delay) {
                max_delay = norm_delay;
                if (cl->ewma_type == TCF_HPD_EWMAR)
                    selected_hold_time = hold_time;
                selected_class = cl;
                selected_delay = class_delay;
            }
        }
        cl = cl->prev_class;
    }

class_selected:

    if (selected_class->q)
        skb = selected_class->q->dequeue(selected_class->q);
    if (skb) {
        sch->q.qlen--;

        if (selected_class->ewma_type == TCF_HPD_EWMAR) {
            if (selected_class->p_samples < selected_class->p_thresh) {
                selected_class->p_samples++;
                selected_class->ewmar_sum += selected_hold_time;
                selected_class->avg_delay = do_div(selected_class->ewmar_sum,
                                                   selected_class->p_samples);
            } else {
                selected_class->avg_delay = selected_delay;
            }
```

```
            if (sch->q.qlen == 0) {
                selected_class->idle_time = now;
            }
        } else {
            selected_class->avg_delay = selected_delay;
        }

        /* Increase time queue index. */
        if (++selected_class->tq_first >= selected_class->tq_length)
            selected_class->tq_first = 0;

        return skb;
    }
    return NULL;
}
```

Appendix B

List of Changed and New Source Files

Linux 2.4 Kernel

File paths are relative to the Linux kernel source root directory.

```
Documentation/Configure.help
include/linux/pkt_sched.h
net/sched/Config.in
net/sched/Makefile
net/sched/sch_api.c
net/sched/sch_hpd.c                    [new]
```

IPRoute2 Package

File paths are relative to the IPRoute2 source root directory.

```
include/linux/pkt_sched.h
tc/Makefile
tc/q_hpd.c                             [new]
```